LAPI Programming Guide
Tenth Edition (November 2009)

This edition applies to:

- version 6, release 1 of IBM AIX for POWER (product number 5765-G62) with the 6100-04-01 Technology Level
- version 5, release 3 of IBM AIX 5L for POWER (product number 5765-G03) with the 5300-07-10 Technology Level
- version 5, release 1 of IBM Parallel Environment for Linux (product number 5765-PEL)

and to all subsequent releases and modifications, until otherwise indicated in new editions. Vertical lines (|) in the left margin indicate technical changes to the previous edition of this book.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

IBM welcomes your comments. A form for your comments appears at the back of this publication. If the form has been removed, address your comments to:
IBM Corporation, Department 58HA, Mail Station P181
2455 South Road
Poughkeepsie, NY 12601-5400
United States of America

FAX (United States and Canada): 1+845+432-9405
FAX (Other Countries)
Your International Access Code +1+845+432-9405

IBMLink (United States customers only): IBMUSM10(MHVRCFS)
Internet: mhvrcfs@us.ibm.com

If you would like a reply, be sure to include your name, address, telephone number, or FAX number.

Make sure to include the following in your comment or note:

- Title and order number of this book
- Page number or topic related to your comment

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
### Requirements

- Hardware .................................................. 35
- Software .................................................. 35

### How is LAPI for Linux packaged?
- Installation steps ........................................ 38
- Uninstallation steps ........................................ 38

### Chapter 6. Setting up, initializing, and terminating LAPI

- Setting and querying the LAPI environment ............ 39
  - Setting environment variables .......................... 39
  - Initializing LAPI ........................................... 42
  - Terminating LAPI .......................................... 44

### Chapter 7. Transferring data

- Data transfer operations ................................... 45
  - Flow of "put" operations ................................ 45
  - Flow of "get" operations ................................ 46
  - Flow of read-modify-write operations ............... 46
- Non-contiguous data transfer ................................. 47
  - Using vectors ............................................. 47
  - Using data gather/scatter programs (DGSPs) ....... 52
- Detecting completion ........................................ 56
  - LAPI handlers ............................................ 56
- LAPI counters ................................................ 61
  - Specifying target-side addresses ...................... 62
  - Additional progress functions ......................... 62
- Bulk message transfer on AIX ................................. 63
- User-initiated RDMA transfer on AIX ....................... 65
  - Resources required for an RDMA job .................. 65
  - RDMA setup ............................................... 66
  - RDMA striping ............................................ 66
  - RDMA considerations ..................................... 66
  - A sample RDMA program ................................ 67
- Using RDMA with the InfiniBand switch ..................... 71

### Chapter 8. Active messaging

- Flow of active message operations ........................ 73
- Using LAPI_Amsend: a complete LAPI program ........... 74

### Chapter 9. Collecting statistics and querying resources

- Printing data transfer statistics ........................ 81
- Querying UDP/IP statistics ................................ 81
- Querying user space statistics ............................. 81
  - For US tasks .............................................. 82
  - Calling a trigger ......................................... 82
- Querying local send statistics ............................. 82
- Querying shared memory statistics ....................... 82
- Querying network resources ................................ 83

### Chapter 10. Using LAPI's profiling interface

- Performing name-shift profiling ........................... 85
- A sample profiling program ................................ 88

### Chapter 11. Compiling LAPI programs

- On an AIX system .......................................... 91
- On a Linux system ........................................... 91
<table>
<thead>
<tr>
<th>Part 3. Advanced LAPI tasks</th>
<th>93</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 12. Advanced programming</td>
<td>95</td>
</tr>
<tr>
<td>The enhanced header handler interface</td>
<td>95</td>
</tr>
<tr>
<td>Setting the ret_flags field</td>
<td>96</td>
</tr>
<tr>
<td>Setting the ctl_flags field</td>
<td>97</td>
</tr>
<tr>
<td>Inline handlers</td>
<td>97</td>
</tr>
<tr>
<td>Optimizing communication for small messages</td>
<td>98</td>
</tr>
<tr>
<td>32-bit and 64-bit interoperability</td>
<td>101</td>
</tr>
<tr>
<td>The lapi_long_t datatype</td>
<td>101</td>
</tr>
<tr>
<td>The LAPI_Address_init64 subroutine</td>
<td>101</td>
</tr>
<tr>
<td>The LAPI_Xfer interface</td>
<td>101</td>
</tr>
<tr>
<td>Chapter 13. LAPI performance considerations</td>
<td>103</td>
</tr>
<tr>
<td>Use of handlers</td>
<td>103</td>
</tr>
<tr>
<td>Running in interrupt mode</td>
<td>103</td>
</tr>
<tr>
<td>Running in UDP/IP mode</td>
<td>104</td>
</tr>
<tr>
<td>Send space and receive space</td>
<td>105</td>
</tr>
<tr>
<td>Running with shared memory enabled</td>
<td>105</td>
</tr>
<tr>
<td>User header data</td>
<td>105</td>
</tr>
<tr>
<td>Send-side copy of small messages</td>
<td>106</td>
</tr>
<tr>
<td>Receive-side optimization for single-packet messages</td>
<td>106</td>
</tr>
<tr>
<td>On-node barrier synchronization</td>
<td>107</td>
</tr>
<tr>
<td>Running multiple LAPI contexts per task</td>
<td>107</td>
</tr>
<tr>
<td>Tunable environment variables</td>
<td>107</td>
</tr>
<tr>
<td>Chapter 14. Lock sharing</td>
<td>111</td>
</tr>
<tr>
<td>Scenarios without lock sharing</td>
<td>111</td>
</tr>
<tr>
<td>Scenarios with lock sharing</td>
<td>115</td>
</tr>
<tr>
<td>Correctness of lock sharing</td>
<td>119</td>
</tr>
<tr>
<td>Implications and restrictions</td>
<td>121</td>
</tr>
<tr>
<td>Initialization and termination</td>
<td>121</td>
</tr>
<tr>
<td>Other LAPI calls</td>
<td>122</td>
</tr>
<tr>
<td>Callbacks</td>
<td>122</td>
</tr>
<tr>
<td>Long critical sections</td>
<td>122</td>
</tr>
<tr>
<td>Lock preemption</td>
<td>122</td>
</tr>
<tr>
<td>Receive/timer interrupts</td>
<td>122</td>
</tr>
<tr>
<td>Performance of multi-threaded programs</td>
<td>123</td>
</tr>
<tr>
<td>Compatibility</td>
<td>123</td>
</tr>
<tr>
<td>A sample lock sharing program</td>
<td>123</td>
</tr>
<tr>
<td>Chapter 15. Striping, failover, and recovery</td>
<td>127</td>
</tr>
<tr>
<td>Using failover and recovery</td>
<td>127</td>
</tr>
<tr>
<td>Monitoring adapter status</td>
<td>127</td>
</tr>
<tr>
<td>Requesting the use of multiple adapters</td>
<td>130</td>
</tr>
<tr>
<td>Failover and recovery restrictions</td>
<td>131</td>
</tr>
<tr>
<td>Data striping</td>
<td>132</td>
</tr>
<tr>
<td>Communication and memory considerations for AIX</td>
<td>133</td>
</tr>
<tr>
<td>IP communication</td>
<td>133</td>
</tr>
<tr>
<td>US communication</td>
<td>134</td>
</tr>
<tr>
<td>Chapter 16. Threaded programming</td>
<td>137</td>
</tr>
<tr>
<td>General guidelines</td>
<td>137</td>
</tr>
<tr>
<td>Using LAPI_Address_init</td>
<td>137</td>
</tr>
<tr>
<td>Making global fence calls</td>
<td>138</td>
</tr>
<tr>
<td>Making &quot;wait on counter&quot; calls</td>
<td>138</td>
</tr>
</tbody>
</table>
Synchronizing threads across tasks .................................. 138
Using handlers ...................................................... 138
LAPI threads ......................................................... 139

Chapter 17. Using LAPI on a standalone system .............. 141
Setting up a standalone system .................................... 141
Initializing a standalone system .................................. 143
  In UDP/IP mode .................................................. 143
  In US mode on AIX ............................................. 143
Compiling LAPI programs on a standalone system .......... 144
Standalone systems: restrictions .................................. 144

Part 4. LAPI reference ............................................. 145

Chapter 18. LAPI commands ....................................... 147
pnsd_stat .......................................................... 148
pnsd_trigger ...................................................... 150

Chapter 19. A sample LAPI subroutine ......................... 153
lapi_subroutines .................................................. 154

Chapter 20. Subroutines for all systems (PE and standalone) .. 157
LAPI_Addr_get ..................................................... 158
LAPI_Addr_set ..................................................... 160
LAPI_Address ...................................................... 162
LAPI_Address_init .............................................. 164
LAPI_Address_init64 ........................................... 166
LAPI_Amsend ...................................................... 168
LAPI_Amsendv ..................................................... 175
LAPI_Fence ......................................................... 181
LAPI_Get .......................................................... 183
LAPI_Getcntr ...................................................... 186
LAPI_Getv .......................................................... 188
LAPI_Gfence ......................................................... 193
LAPI_Group_create ............................................. 195
LAPI_Group_free .................................................. 197
LAPI_Init .......................................................... 199
LAPI_Msg_string .................................................... 206
LAPI_Msgpoll ...................................................... 208
LAPI_Probe ........................................................ 211
LAPI_Put ........................................................... 213
LAPI_Putv .......................................................... 216
LAPI_Qenv .......................................................... 221
LAPI_Rmw .......................................................... 226
LAPI_Rmw64 ......................................................... 230
LAPI_Senv .......................................................... 234
LAPI_Setcntr ......................................................... 236
LAPI_Term ........................................................... 239
LAPI_Util ............................................................ 241
LAPI_Waitcntr ...................................................... 258
LAPI_Xfer ........................................................... 260

Chapter 21. Subroutines for standalone systems .............. 281
LAPI_Nopoll_wait ................................................... 282
LAPI_Purge_totask ............................................... 284
LAPI_Resume_totask .............................................. 286
Chapter 22. Sample LAPI programs ........................................... 291
Sample program directory structure ...................................... 291
Using the LAPI sample programs ........................................... 296
Summary of constructs and techniques for LAPI programming .... 296

Appendix A. Accessibility features for RSCT ................................. 299
Accessibility features ......................................................... 299
Related accessibility information ........................................... 299
IBM and accessibility ....................................................... 299

Appendix B. Product-related information .................................... 301
LAPI version ................................................................. 301
Case sensitivity in AIX ...................................................... 302
ISO 9000 ........................................................................ 302
Product-related feedback ..................................................... 302

Appendix C. LAPI execution models ........................................... 303
The IP/US execution model .................................................. 303
The shared memory execution model ..................................... 305
  LAPI shared memory ....................................................... 306

Appendix D. LAPI messages, return codes, and return values .......... 309
LAPI attention messages .................................................... 309
LAPI return codes ........................................................... 309
LAPI error codes ............................................................ 309
LAPI return values .......................................................... 313

Appendix E. LAPI environment variables and runtime attributes ........ 317
Environment variables ....................................................... 317
  Variables for communication ........................................... 317
  Variables for data transfer .............................................. 317
  Variables for diagnostics .............................................. 319
  Variables for multicasting ............................................. 319
  Variables for performance tuning ................................... 320
  Variables for POE ......................................................... 321
  Variables for shared memory ......................................... 323
  Variables for standalone systems .................................... 324
Runtime attributes .......................................................... 325
  Attributes you can query or set ...................................... 325
  Attributes you can query ............................................... 325

Appendix F. LAPI datatypes ...................................................... 329

Appendix G. LAPI constants and size limits ................................ 333

Appendix H. LAPI restrictions .................................................. 335
General restrictions .......................................................... 335
Use of segment registers on AIX .......................................... 335
Other restrictions ............................................................ 335

Notices .......................................................................... 337
Trademarks .................................................................... 338

Glossary ........................................................................ 341
Index .......................................... 345
# Figures

1. How LAPI sets up the mode of communication ........................................ 42
2. The sequence of events for a "put" operation ........................................ 46
3. The sequence of events for a "get" operation ........................................ 46
4. The sequence of events for a read-modify-write operation ....................... 47
5. Transferring data with type LAPI_GEN_IOVECTOR .................................. 50
6. Transferring data with type LAPI_GEN_STRIDED_XFER ............................ 51
7. The completion handler queue .............................................................. 60
8. Packet data flow ..................................................................................... 64
9. Bulk data flow .......................................................................................... 65
10. The sequence of events for an active message operation ....................... 74
11. Inline completion handler flow ............................................................... 98
12. A program initiates a call to LAPI, without lock sharing ....................... 112
13. LAPI initiates a callback, without lock sharing ....................................... 113
14. A program initiates a call to LAPI, with embedded up- and down-calls .. 114
15. LAPI initiates a callback, with embedded up- and down-calls .................. 115
16. A program initiates a call to LAPI, with one lock acquisition and release pair 116
17. LAPI initiates a callback, with one lock acquisition and release pair ....... 117
18. A program initiates a call to LAPI, with embedded up- and down-calls and one lock acquisition and release pair .............................................. 118
19. LAPI initiates a callback, with embedded up- and down-calls and one lock acquisition and release pair .................................................. 119
20. Critical sections under lock sharing ......................................................... 121
21. Execution sequence of the accumulate_and_return.Am sample ............ 294
22. Execution sequence of the accumulate_and_return.Put sample ............. 294
23. A LAPI thread model .............................................................................. 304
24. LAPI_Put without shared memory .......................................................... 306
25. LAPI_Put with shared memory ............................................................... 306
# Tables

1. Conventions ................................ xiii
2. Terminology ................................ xv
3. Changes in this edition ............................. 3
5. Changes in the eighth edition ........................... 4
6. Changes in the seventh edition ........................... 5
7. Changes in the sixth edition ........................... 5
8. Changes in the fifth edition ........................... 5
9. Changes in the fourth edition ........................... 6
10. Changes in the third edition ........................... 6
11. Changes in the second edition .......................... 7
12. Changes in the first edition ........................... 7
13. Differences among LAPI versions ........................ 9
14. LAPI RPMs required for installation ..................... 36
15. Directories and files associated with the LAPI RPMs ..................... 37
16. Rules for vector transfer .............................. 52
17. LAPI handlers ................................ 61
18. LAPI counters ................................ 62
19. Blocks of rCxts per task that are required for each window for an RDMA job ............. 67
20. Total blocks of rCxts per adapter required with multiple tasks sharing an adapter ........ 67
21. LAPI profiling interfaces ........................................ 85
22. Compiling LAPI programs on a system running PE for AIX ............................. 91
23. Compiling LAPI programs on a system running PE for Linux ......................... 91
24. Failover and recovery operations on AIX ..................... 131
25. Failover and recovery operations on Linux .................... 131
26. Compiling RSCT LAPI for AIX programs on a standalone system ...................... 144
27. Compiling LAPI for Linux programs on a standalone system .......................... 144
28. lapi_util_type_t types ............................................. 242
29. The lapi_reg_dgsp_t fields ................................. 243
30. The lapi_resv_dgsp_t fields ................................. 243
31. The lapi_dref_dgsp_t fields ................................. 244
32. The lapi_reg_ddm_t fields ................................. 245
33. The lapi_pack_dgsp_t fields ................................. 245
34. The lapi_unpack_dgsp_t fields ................................. 246
35. The lapi_add_udp_port_t fields ................................. 246
36. The lapi_add_udp_port_ext fields ................................. 247
37. The lapi_thread_func_t fields ................................. 247
38. The lapi_get_pvo_t fields ................................. 251
39. The lapi_rdma_notification_t fields ................................. 252
40. The lapi_remote_cxt_t fields ................................. 253
41. The lapi_trigger_util_t fields ................................. 253
42. The lapi_trigger_util_t fields ................................. 253
43. LAPI_Xfer structure types ................................. 261
44. LAPI_Amsend and lapi_am_t equivalents ..................... 262
45. The lapi_amdgsp_t fields ................................. 263
46. The lapi_am_t fields for LAPI_AM_LW_XFER .......................... 264
47. LAPI_Amsendv and lapi_amv_t equivalents ................... 265
48. The lapi_amx_t fields ................................. 266
49. LAPI_Get and lapi_get_t equivalents ...................... 267
50. LAPI_Getv and lapi_getv_t equivalents .................... 268
51. LAPI_Put and lapi_put_t equivalents ...................... 269
52. LAPI_Putv and lapi_putv_t equivalents .................... 270
53. The lapi_hwxfer_t fields ................................. 271
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>54. LAPI_Rmw and lapi_rmw_t equivalents</td>
<td>272</td>
</tr>
<tr>
<td>55. The lapi_mc_t fields</td>
<td>274</td>
</tr>
<tr>
<td>56. Constructs and techniques for LAPI programming</td>
<td>296</td>
</tr>
<tr>
<td>57. LAPI attention messages</td>
<td>309</td>
</tr>
<tr>
<td>58. LAPI return codes</td>
<td>309</td>
</tr>
<tr>
<td>59. LAPI error codes</td>
<td>310</td>
</tr>
<tr>
<td>60. LAPI return values</td>
<td>313</td>
</tr>
<tr>
<td>61. Environment variables for communication</td>
<td>317</td>
</tr>
<tr>
<td>62. Environment variables for data transfer</td>
<td>317</td>
</tr>
<tr>
<td>63. Environment variables for diagnostics</td>
<td>319</td>
</tr>
<tr>
<td>64. Environment variables for multicasting</td>
<td>320</td>
</tr>
<tr>
<td>65. Environment variables for performance tuning.</td>
<td>320</td>
</tr>
<tr>
<td>66. Environment variables for POE</td>
<td>322</td>
</tr>
<tr>
<td>67. Environment variables for shared memory</td>
<td>323</td>
</tr>
<tr>
<td>68. Environment variables for standalone systems</td>
<td>324</td>
</tr>
<tr>
<td>69. Runtime attributes you can query or set</td>
<td>325</td>
</tr>
<tr>
<td>70. Attributes that return integers</td>
<td>325</td>
</tr>
<tr>
<td>71. Attributes that return multiple values</td>
<td>326</td>
</tr>
<tr>
<td>72. LAPI datatypes</td>
<td>329</td>
</tr>
</tbody>
</table>
About this book

This book includes conceptual, procedural, and reference information about the low-level application programming interface (LAPI). This book describes two implementations of LAPI:
1. the AIX version that is a component of Reliable Scalable Cluster Technology (RSCT) for AIX 5.3 and 6.1
2. the Linux version, which is shipped with Parallel Environment (PE) for Linux

Differences among the various implementations of LAPI are noted in Table 13 on page 9 and throughout the book, where applicable.

Who should use this book

This book is intended for programmers who want to write and run LAPI programs on the AIX® or Linux operating system. The programmer should be experienced with UNIX®-like environments, networked systems, and the C, C++, or FORTRAN programming language.

Conventions and terminology used in this information

This information includes the following conventions and terminology.

Conventions

Table 1 shows the conventions used in this information:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>bold</td>
<td><strong>Bold</strong> words or characters represent system elements that you must use literally, such as commands, flags, path names, directories, file names, values, and selected menu options.</td>
</tr>
<tr>
<td>bold underlined</td>
<td><strong>Bold underlined</strong> keywords are defaults. These take effect if you do not specify a different keyword.</td>
</tr>
<tr>
<td>constant width</td>
<td>Examples and information that the system displays appear in constant-width typeface.</td>
</tr>
<tr>
<td>italic</td>
<td><strong>Italic</strong> words or characters represent variable values that you must supply. <strong>Italics</strong> are also used for information unit titles, for the first use of a glossary term, and for general emphasis in text.</td>
</tr>
</tbody>
</table>
Table 1. Conventions (continued)

<table>
<thead>
<tr>
<th>Convention</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>underlined</td>
<td>1. When used to show the size of a parameter, a comparison of values, or a range of values, valid values for the query parameter of the LAPI_Qenv subroutine are underlined. For example:</td>
</tr>
<tr>
<td></td>
<td><strong>NUM_TASKS</strong> is a shorthand notation for:</td>
</tr>
<tr>
<td></td>
<td>LAPI_Qenv(hndl, <strong>NUM_TASKS</strong>, ret_val)</td>
</tr>
<tr>
<td></td>
<td>For a list of the query parameter's valid values, see LAPI_Qenv on page 221.</td>
</tr>
<tr>
<td></td>
<td>2. Underlined characters are also a shorthand notation for:</td>
</tr>
<tr>
<td></td>
<td>LAPI_Xfer with transfer type LAPI_xfer-type_XFER</td>
</tr>
<tr>
<td></td>
<td>The valid values for xfer-type are: LAPI_AM_XFER, LAPI_AM_LW_XFER, LAPI_AMV_XFER, LAPI_AMX_XFER, LAPI_DGSP_XFER, LAPI_GET_XFER, LAPI_GETV_XFER, LAPI_MC_XFER, LAPI_PUT_XFER, LAPI_PUTV_XFER, LAPI_RDMA_XFER, and LAPI_RMW_XFER.</td>
</tr>
<tr>
<td></td>
<td>So, for example, this document would refer to:</td>
</tr>
<tr>
<td></td>
<td>LAPI_Xfer with transfer type LAPI_AM_XFER</td>
</tr>
<tr>
<td></td>
<td>as:</td>
</tr>
<tr>
<td></td>
<td>AM</td>
</tr>
<tr>
<td></td>
<td>For more information about these transfer types, see LAPI_Xfer on page 260.</td>
</tr>
<tr>
<td>&lt;key&gt;</td>
<td>Angle brackets (less-than and greater-than) enclose the name of a key on the keyboard. For example, &lt;Enter&gt; refers to the key on your terminal or workstation that is labeled with the word Enter.</td>
</tr>
<tr>
<td>\</td>
<td>In command examples, a backslash indicates that the command or coding example continues on the next line. For example: mkcondition -r IBM.FileSystem -e &quot;PercentTotUsed &gt; 90&quot; \ -E &quot;PercentTotUsed &lt; 85&quot; -m p &quot;FileSystem space used&quot;</td>
</tr>
<tr>
<td>{item}</td>
<td>Braces enclose a list from which you must choose an item in format and syntax descriptions.</td>
</tr>
<tr>
<td>[item]</td>
<td>Brackets enclose optional items in format and syntax descriptions.</td>
</tr>
<tr>
<td>&lt;Ctrl-x&gt;</td>
<td>The notation &lt;Ctrl-x&gt; indicates a control character sequence. For example, &lt;Ctrl-c&gt; means that you hold down the control key while pressing &lt;c&gt;.</td>
</tr>
<tr>
<td>item...</td>
<td>Ellipses indicate that you can repeat the preceding item one or more times.</td>
</tr>
<tr>
<td></td>
<td>• In synopsis statements, vertical lines separate a list of choices. In other words, a vertical line means Or.</td>
</tr>
<tr>
<td></td>
<td>• In the left margin of the document, vertical lines indicate technical changes to the information.</td>
</tr>
</tbody>
</table>
Terminology

Table 2 includes some of the terminology used in this information:

<table>
<thead>
<tr>
<th>Term</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPS</td>
<td>A shorthand notation for the High Performance Switch, which works in conjunction with a specific family of IBM® System p® servers.</td>
</tr>
</tbody>
</table>

See the "Glossary" on page 341 for definitions of some of the other terms that are used in this publication.

Prerequisite and related information

The Reliable Scalable Cluster Technology (RSCT) library includes the following publications:

- **RSCT: Administration Guide**, SA22-7889, provides an overview of the RSCT components and describes how to:
  - Create and administer RSCT peer domains.
  - Manage and monitor resources using the resource monitoring and control (RMC) subsystem.
  - Administer cluster security services for RSCT peer domains and CSM management domains.

- **RSCT: Diagnosis Guide**, SA23-2202, describes how to diagnose and resolve problems related to the various components of RSCT. This book is a companion volume to **RSCT: Messages**, which lists the error messages that may be generated by each RSCT component. While **RSCT: Messages** describes the appropriate user responses to messages that are generated by RSCT components, this book contains additional and more detailed diagnostic procedures.

- **RSCT: Group Services Programming Guide and Reference**, SA22-7888, contains information for programmers who want to write new clients that use the group services subsystem’s application programming interface (GSAPI) or who want to add the use of group services to existing programs. This book is intended for programmers of system management applications who want to use group services to make their applications highly available.

- **RSCT: LAPI Programming Guide**, SA22-7936, provides conceptual, procedural, and reference information about the low-level application programming interface (LAPI). LAPI, a message-passing API that is based on an “active message style” mechanism, provides a one-sided communication model and optimal performance across various high-performance switches on AIX and Linux systems.

- **RSCT for AIX: Managing Shared Disks**, SA22-7937, describes the shared disk management facilities of IBM eServer™ Cluster 1600 server processors — the optional virtual shared disk and recoverable virtual shared disk components of RSCT for AIX. This book describes how you can use these components to manage cluster disks to enable multiple nodes to share the information they hold. The book includes an overview of the components and explains how to plan for them, install them, and use them to add reliability and availability to your data storage.

- **RSCT: Messages**, GA22-7891, lists the error messages that may be generated by each RSCT component. For each message, this manual provides an explanation of the message, and describes how you should respond to it.
• RSCT: NRT API Programming Guide, SC23-6655, contains information for programmers who want to use the network resource table (NRT) API to work with adapter window resources on Infiniband systems.

• RSCT: RMC Programming Guide and Reference, SA23-1346, describes the resource monitoring and control application programming interface (RMC API). This book is intended for programmers who want to create applications that use the RMC API to connect to the RMC subsystem to leverage its resource management and monitoring capabilities.

• RSCT for AIX: Technical Reference, SA22-7890, and RSCT for Multiplatforms: Technical Reference, SA22-7893, provide detailed reference information about all of the RSCT commands, daemons, files, and scripts.

For access to all of the RSCT documentation, go to the IBM Cluster information center at:

http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp

This Web site contains the most recent RSCT documentation in HTML and PDF formats. The Cluster information center also includes an RSCT Documentation Updates file, which contains documentation corrections and clarifications, as well as information that was discovered after the RSCT books were published. Check this file for pertinent information (about required software patches, for example).

The current RSCT books and earlier versions of the library are also available in PDF format from the IBM Publications Center Web site, which is located at:

http://www.ibm.com/shop/publications/order

It is easiest to locate a manual in the IBM Publications Center by supplying the manual's publication number. The publication number for each of the RSCT books is listed after the book title in the preceding list.

How to send your comments

Your feedback is important in helping to provide accurate, high-quality information. If you have any comments about this book or any other RSCT documentation:

• Go to the IBM Cluster information center home page at:
  http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp
  Click on the Contact us link to go to our feedback page, where you can enter and submit your comments.

• Send your comments by e-mail to: mhvrcfs@us.ibm.com
  Include the book title and order number, and, if applicable, the specific location of the information about which you have comments (for example, a page number, table number, or figure number).

• Fill out one of the forms at the back of this book and return it by mail, by fax, or by giving it to an IBM representative.
**Part 1. LAPI concepts**

<table>
<thead>
<tr>
<th>Chapter 1. What’s new in LAPI?</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tips for LAPI users</td>
<td>8</td>
</tr>
<tr>
<td>Migration and coexistence considerations</td>
<td>9</td>
</tr>
</tbody>
</table>

| Chapter 2. What is the low-level application programming interface (LAPI)? | 11 |
| Why use LAPI? | 15 |

| Chapter 3. An overview of LAPI | 17 |
| Initialization and termination | 18 |
| Querying and setting up the runtime environment | 18 |
| Address-related functions | 19 |
| Put and get functions | 19 |
| Active messages | 20 |
| Non-contiguous data transfer | 21 |
| Remote read-modify-write functions | 21 |
| Generic data transfer | 22 |
| Extended user header support | 23 |
| Multicasting | 23 |
| Low-latency API extensions | 23 |
| User-initiated RDMA transfer | 24 |
| Utility functions | 24 |
| Monitoring progress | 24 |
| Message ordering | 25 |
| Error message functions | 26 |
| Recovery-related functions | 26 |
Chapter 1. What’s new in LAPI?

LAPI for Linux and RSCT LAPI for AIX include several enhancements. To take advantage of some of these enhancements, you might need to make changes to your existing LAPI programs. See “Tips for LAPI users” on page 8 for more information.

Major changes and additions to LAPI include:

Table 3. Changes in this edition

<table>
<thead>
<tr>
<th>Change or addition</th>
<th>For more information, see:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for IP Version 6, which includes a new data structure (in C): <code>lapi_add_udp_port_ext</code></td>
<td>&quot;LAPI_ADD_UDP_DEST_EXT&quot; on page 246</td>
</tr>
<tr>
<td>Deprecated data structure (in C): <code>lapi_add_udp_port_t</code></td>
<td></td>
</tr>
<tr>
<td><strong>On RSCT LAPI for AIX:</strong></td>
<td></td>
</tr>
<tr>
<td>Support for multiple parallel APIs.</td>
<td>&quot;Setting environment variables&quot; on page 39</td>
</tr>
<tr>
<td>The ability to query network resources.</td>
<td>&quot;Querying network resources&quot; on page 83</td>
</tr>
<tr>
<td>A new PNSD configuration file (PNSD.cfg).</td>
<td>&quot;The PNSD component of LAPI&quot; on page 128</td>
</tr>
<tr>
<td>Changed subroutines: <code>LAPI_Addr_set</code>, <code>LAPI_Init</code>, <code>LAPI_Qenv</code>.</td>
<td>&quot;LAPI_Addr_set&quot; on page 160</td>
</tr>
<tr>
<td></td>
<td>&quot;LAPI_Init&quot; on page 199</td>
</tr>
<tr>
<td></td>
<td>&quot;LAPI_Qenv&quot; on page 221</td>
</tr>
<tr>
<td>Changed datatypes: <code>lapi_info_t</code>, <code>lapi_query_t</code>.</td>
<td>&quot;LAPI_Init&quot; on page 199</td>
</tr>
<tr>
<td></td>
<td>&quot;LAPI_Qenv&quot; on page 221</td>
</tr>
<tr>
<td>Changed environment variables: <code>MP_INFOLEVEL</code>, <code>MP_MSG_API</code>, <code>MP_RC_MAX_QP</code>.</td>
<td>Appendix E, &quot;LAPI environment variables and runtime attributes,&quot; on page 317</td>
</tr>
<tr>
<td>Deprecated environment variable: <code>MP_RDMA_COUNT</code>.</td>
<td></td>
</tr>
<tr>
<td>Deprecated error code: 407.</td>
<td></td>
</tr>
<tr>
<td>Deprecated return value: <code>LAPI_ERR_NO_NETSTR_SET</code>.</td>
<td></td>
</tr>
<tr>
<td><strong>On LAPI for Linux:</strong></td>
<td></td>
</tr>
<tr>
<td>Support for the SLES 11 operating system.</td>
<td>&quot;LAPI_Addr_set&quot; on page 160</td>
</tr>
<tr>
<td></td>
<td>&quot;LAPI_Init&quot; on page 199</td>
</tr>
<tr>
<td></td>
<td>&quot;LAPI_Qenv&quot; on page 221</td>
</tr>
<tr>
<td></td>
<td>Appendix E, &quot;LAPI environment variables and runtime attributes,&quot; on page 317</td>
</tr>
</tbody>
</table>

Table 4. Changes in the ninth edition

<table>
<thead>
<tr>
<th>Change or addition</th>
<th>For more information, see:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for the AIX 6.1 operating system, when running in user space (US) mode.</td>
<td></td>
</tr>
<tr>
<td>With LAPI for Linux, support for the Red Hat EL 5 operating system, when running in UDP/IP mode.</td>
<td>Table 14 on page 36</td>
</tr>
<tr>
<td>With LAPI for Linux (over UDP/IP) and RSCT LAPI for AIX 6.1 on InfiniBand systems, support for:</td>
<td></td>
</tr>
<tr>
<td>• checkpoint and restart operations.</td>
<td></td>
</tr>
<tr>
<td>• job preemption.</td>
<td></td>
</tr>
<tr>
<td>• remote direct memory access (RDMA) communication.</td>
<td></td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 2003, 2009
Table 4. Changes in the ninth edition (continued)

<table>
<thead>
<tr>
<th>Change or addition</th>
<th>For more information, see:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for multicasting. This support includes:</td>
<td></td>
</tr>
<tr>
<td>• New subroutines: <strong>LAPI_Group_create</strong> and <strong>LAPI_Group_free</strong>.</td>
<td><img src="#" alt="LaTeX image" /></td>
</tr>
<tr>
<td>• A new <strong>LAPI_Xfer</strong> transfer type — <strong>LAPI_MC_XFER</strong> (MC) and its corresponding datatype — <strong>lapi_mc_t</strong>.</td>
<td><img src="#" alt="LaTeX image" /></td>
</tr>
<tr>
<td>• A new datatype: <strong>lapi_group_t</strong>.</td>
<td><img src="#" alt="LaTeX image" /></td>
</tr>
<tr>
<td>The ability to query US statistics and call triggers. This support includes:</td>
<td><img src="#" alt="LaTeX image" /></td>
</tr>
<tr>
<td>• New commands: <strong>pnsd_stat</strong> and <strong>pnsd_trigger</strong>.</td>
<td><img src="#" alt="LaTeX image" /></td>
</tr>
<tr>
<td>• New <strong>LAPI_Util</strong> functions: <strong>LAPI_TRIGGER_ADD</strong> and <strong>LAPI_TRIGGER_REMOVE</strong> and corresponding datatypes — <strong>lapi_trigger_function_t</strong>.</td>
<td><img src="#" alt="LaTeX image" /></td>
</tr>
<tr>
<td>With RSCT LAPI for AIX 6.1, support for the barrier synchronization register (BSR).</td>
<td><img src="#" alt="LaTeX image" /></td>
</tr>
<tr>
<td>LAPI now allows inline transfers to be sent out in header handlers, in addition to receive completion handlers.</td>
<td><img src="#" alt="LaTeX image" /></td>
</tr>
<tr>
<td>The <strong>LAPI_DROP_PKT</strong> flag in the <strong>ctl_flags</strong> field of the <strong>lapi_return_info_t</strong> structure has been removed.</td>
<td><img src="#" alt="LaTeX image" /></td>
</tr>
<tr>
<td>With LAPI for Linux, support for Red Hat EL 4 and SLES9 has been removed.</td>
<td><img src="#" alt="LaTeX image" /></td>
</tr>
</tbody>
</table>

Table 5. Changes in the eighth edition

<table>
<thead>
<tr>
<th>Change or addition</th>
<th>For more information, see:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for the AIX 6.1 operating system, when running in UDP/IP mode.</td>
<td><img src="#" alt="LaTeX image" /></td>
</tr>
<tr>
<td>RSCT LAPI for AIX 6.1 includes all of the same features and functions as RSCT LAPI for AIX 5.3.</td>
<td><img src="#" alt="LaTeX image" /></td>
</tr>
<tr>
<td>With RSCT LAPI for AIX 5.3 on InfiniBand systems, support for:</td>
<td><img src="#" alt="LaTeX image" /></td>
</tr>
<tr>
<td>• checkpoint and restart operations.</td>
<td><img src="#" alt="LaTeX image" /></td>
</tr>
<tr>
<td>• job preemption.</td>
<td><img src="#" alt="LaTeX image" /></td>
</tr>
<tr>
<td>• remote direct memory access (RDMA) communication.</td>
<td><img src="#" alt="LaTeX image" /></td>
</tr>
<tr>
<td>This support includes two new environment variables: <strong>MP_RC_MAX_QP</strong> and <strong>MP_RC_USE_LMC</strong>.</td>
<td><img src="#" alt="LaTeX image" /></td>
</tr>
<tr>
<td>The <strong>rsct.lapi.nam</strong> fileset is no longer required to support failover and recovery. Failover and recovery functions are now handled in the LAPI protocol library.</td>
<td><img src="#" alt="LaTeX image" /></td>
</tr>
<tr>
<td>RSCT LAPI for AIX 5.2 is functionally stabilized at LAPI version 2.3.3.0, so all references to LAPI support on AIX 5.2 have been removed from this document.</td>
<td><img src="#" alt="LaTeX image" /></td>
</tr>
<tr>
<td>LAPI is no longer supported on Parallel System Support Programs (PSSP), so all references to LAPI support on PSSP have been removed from this document.</td>
<td><img src="#" alt="LaTeX image" /></td>
</tr>
</tbody>
</table>
## Table 6. Changes in the seventh edition

<table>
<thead>
<tr>
<th>Change or addition</th>
<th>For more information, see:</th>
</tr>
</thead>
<tbody>
<tr>
<td>With RSCT LAPI for AIX 5.3, support for:</td>
<td>&quot;The PNSD component of LAPI&quot; on page 128</td>
</tr>
<tr>
<td>- the protocol network services daemon (PNSD)</td>
<td>RSCT: NRT API Programming Guide, SC23-6655</td>
</tr>
<tr>
<td>- the InfiniBand switch</td>
<td></td>
</tr>
<tr>
<td>Support for the network resource table (NRT) API</td>
<td></td>
</tr>
</tbody>
</table>

## Table 7. Changes in the sixth edition

<table>
<thead>
<tr>
<th>Change or addition</th>
<th>For more information, see:</th>
</tr>
</thead>
<tbody>
<tr>
<td>With RSCT LAPI for AIX 5.3 and LAPI for Linux:</td>
<td>&quot;Extended user header support&quot; on page 23</td>
</tr>
<tr>
<td>- extended user header support. This support includes:</td>
<td>&quot;lapi_amx_t details&quot; on page 266</td>
</tr>
<tr>
<td>- A new <strong>LAPI_Xfer</strong> transfer type — <strong>LAPI_AMX_XFER (AMX)</strong>, which allows user headers that are up to 2 GB in length, and its corresponding datatype — <strong>lapi_amx_t</strong>.</td>
<td>&quot;The enhanced header handler interface&quot; on page 95</td>
</tr>
<tr>
<td>- A new field in the <strong>lapi_return_info_t</strong> structure — <strong>recv_offset_dgps_bytes</strong> — which only applies when <strong>AMX</strong> is used.</td>
<td></td>
</tr>
<tr>
<td>New tunables for performance when running in UDP/IP mode.</td>
<td>&quot;Running in UDP/IP mode&quot; on page 104</td>
</tr>
<tr>
<td>With LAPI for Linux, support for the SLES 10 operating system.</td>
<td>Table 14 on page 36</td>
</tr>
</tbody>
</table>

## Table 8. Changes in the fifth edition

<table>
<thead>
<tr>
<th>Change or addition</th>
<th>For more information, see:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Linux implementation of LAPI, which includes almost all of the same features and functions as RSCT LAPI for AIX 5L. Here are the major differences:</td>
<td>Table 13 on page 9</td>
</tr>
<tr>
<td>- LAPI for Linux supports the InfiniBand switch, rather than the IBM High Performance Switch (HPS)</td>
<td>Chapter 5, “Installing LAPI for Linux,” on page 35</td>
</tr>
<tr>
<td>- For failover and recovery, LAPI for Linux uses the protocol network services daemon (PNSD), rather than the network availability matrix (NAM) pseudo-device</td>
<td>&quot;The PNSD component of LAPI&quot; on page 128</td>
</tr>
<tr>
<td>- LAPI for Linux does not include the shared memory kernel extension or support for:</td>
<td>&quot;Variables for communication&quot; on page 317</td>
</tr>
<tr>
<td>- bulk message transfer</td>
<td></td>
</tr>
<tr>
<td>- checkpoint and restart operations</td>
<td></td>
</tr>
<tr>
<td>- data striping</td>
<td></td>
</tr>
<tr>
<td>- job preemption</td>
<td></td>
</tr>
<tr>
<td>- user-initiated RDMA transfer</td>
<td></td>
</tr>
<tr>
<td>A new environment variable: <strong>MP_DEVTYPE</strong></td>
<td></td>
</tr>
</tbody>
</table>
### Table 9. Changes in the fourth edition

<table>
<thead>
<tr>
<th>Change or addition</th>
<th>For more information, see:</th>
</tr>
</thead>
<tbody>
<tr>
<td>For AIX 5.3, support for user-initiated RDMA transfer using RDMA operations. This support includes:</td>
<td></td>
</tr>
<tr>
<td>1. Two new runtime attributes that let LAPI clients get information related to RDMA:</td>
<td></td>
</tr>
<tr>
<td>RDMA_REMOTE_RCXT_AVAIL and RDMA_REMOTE_RCXT_TOTAL.</td>
<td></td>
</tr>
<tr>
<td>2. Three new LAPI_Util functions:</td>
<td></td>
</tr>
<tr>
<td>LAPI_REGISTER_NOTIFICATION,</td>
<td></td>
</tr>
<tr>
<td>LAPI_REMOTE_RCXT, and</td>
<td></td>
</tr>
<tr>
<td>LAPI_XLATE_ADDRESS and their corresponding datatypes — lapi_rdma_notification_t,</td>
<td></td>
</tr>
<tr>
<td>lapi_remote_cxt_t, and lapi_get_pvo_t.</td>
<td></td>
</tr>
<tr>
<td>3. A new LAPI_Xfer transfer type — LAPI_RDMA_XFER (RDMA) and its corresponding</td>
<td></td>
</tr>
<tr>
<td>datatype — lapi_hwxfer_t.</td>
<td></td>
</tr>
<tr>
<td>4. Two new environment variables: MP_RCXT_BLKS and MP_RDMA_COUNT</td>
<td></td>
</tr>
<tr>
<td>5. Additional new datatypes.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>For AIX 5.3, support for low-latency API extensions. This support includes a new LAPI Xfer transfer type for small messages: LAPI_AM_LW_XFER (AM_LW).</td>
<td></td>
</tr>
</tbody>
</table>

### Table 10. Changes in the third edition

<table>
<thead>
<tr>
<th>Change or addition</th>
<th>For more information, see:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for the IBM High Performance Switch (HPS) for a specific family of IBM System p servers.</td>
<td>Cluster 1600 High Performance Switch Planning, Installation, and Service for p5 Servers</td>
</tr>
<tr>
<td>To use the HPS, you need to have version 1.4.1 (or later) of Cluster Systems Management (CSM) for AIX 5L™ (product number 5765-F67) installed on your system.</td>
<td>CSM for AIX 5L and Linux: Planning and Installation Guide</td>
</tr>
<tr>
<td>Striping support.</td>
<td>Data striping” on page 132</td>
</tr>
<tr>
<td>Bulk transfer of messages using the RDMA protocol. This mechanism can only be used in conjunction with the HPS.</td>
<td>Bulk message transfer on AIX” on page 63</td>
</tr>
<tr>
<td>This support includes two new runtime attributes that let LAPI clients get information related to bulk transfer (BULK_XFER and BULK_MIN_MSG_SIZE).</td>
<td></td>
</tr>
<tr>
<td>Two additional runtime attributes that let LAPI clients get information about new statistics (QUERY_LOCAL_SEND_STATISTICS and QUERY_SHM_STATISTICS)</td>
<td>Chapter 9, “Collecting statistics and querying resources,” on page 81</td>
</tr>
<tr>
<td>The maximum number of shared memory tasks per operating system image has changed from 32 to 128.</td>
<td>Requirements and considerations” on page 307</td>
</tr>
<tr>
<td>Support for 8192 user space (US) MPI tasks in a single job.</td>
<td></td>
</tr>
</tbody>
</table>
Table 11. Changes in the second edition

<table>
<thead>
<tr>
<th>Change or addition</th>
<th>For more information, see:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failover and recovery support.</td>
<td>“Using failover and recovery” on page 127</td>
</tr>
</tbody>
</table>
| Lock sharing support, which includes a new LAPI utility for shared locking and signalling functions. | • Chapter 14, “Lock sharing,” on page 111  
  • “LAPI_GET_THREAD_FUNC” on page 247 |
| Environment variable changes.                                                      | “Variables for performance tuning” on page 320 |
| New: MP_INSTANCES, MP_LAPI_INET_ADDR, MP_RETRANSMIT_INTERVAL, MP_REXMIT_BUF_CNT, MP_REXMIT_BUF_SIZE. | |
| Deprecated: MP_COPY_SEND_BUF_SIZE.                                                 | |

Table 12. Changes in the first edition

<table>
<thead>
<tr>
<th>Change or addition</th>
<th>For more information, see:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for the IBM eServer pSeries® High Performance Switch (pSeries HPS).</td>
<td>eServer Cluster 1600: pSeries High Performance Switch Planning, Installation, and Service</td>
</tr>
<tr>
<td>To use the pSeries HPS, you need to have version 1.3.2 (or later) of Cluster Systems Management (CSM) for AIX 5L (product number 5765-F67) installed on your system.</td>
<td>CSM for AIX 5L and Linux: Planning and Installation Guide</td>
</tr>
<tr>
<td>The ability to run over the user datagram protocol (UDP) rather than the user space (US) protocol.</td>
<td>“LAPI communication modes” on page 41</td>
</tr>
<tr>
<td>Inline completion handlers, which allow your completion handler to be called from within the thread of execution that completed the data transfer. For applications that rely on completion handlers rather than counters, this can provide a significant performance enhancement.</td>
<td>“Inline handlers” on page 97</td>
</tr>
<tr>
<td>A generalized mechanism to transfer arbitrary portions of non-contiguous data using data gather/scatter programs (DGSPs).</td>
<td>“Using data gather/scatter programs (DGSPs)” on page 52</td>
</tr>
<tr>
<td>Runtime attributes that let LAPI clients control the communication library and get information about new statistics (PRINT_STATISTICS and QUERY_STATISTICS)</td>
<td>“LAPI_Qenv” on page 221 and Attributes that return multiple values” on page 326</td>
</tr>
<tr>
<td>A polling function that provides more flexible access to invoking the LAPI dispatcher explicitly and lets you poll for messages without polling for a specific counter.</td>
<td>“LAPI_Msgpoll” on page 208</td>
</tr>
<tr>
<td>Support for multi-threaded programs. LAPI calls can be made from multiple user threads.</td>
<td>Chapter 16, “Threaded programming,” on page 137</td>
</tr>
<tr>
<td>This mechanism can only be used in conjunction with the HPS or the pSeries HPS.</td>
<td>“Bulk message transfer on AIX” on page 63</td>
</tr>
<tr>
<td>An API call that performs various LAPI utility operations, most notably for user DGSPs.</td>
<td>“LAPI_Util” on page 241</td>
</tr>
<tr>
<td>A LAPI_Xfer transfer type to support DGSP transfers: LAPI_DGSP_XFER (DGSP).</td>
<td>“lapi_amdgsp_t details” on page 262</td>
</tr>
</tbody>
</table>
Table 12. Changes in the first edition (continued)

<table>
<thead>
<tr>
<th>Change or addition</th>
<th>For more information, see:</th>
</tr>
</thead>
</table>
| Several new data structures. | • For extended header handler support:  
| | lapi_return_info_t (see "LAPI_Amsend" on page 168)  
| | • For message polling: lapi_msg_info_t (see "LAPI_Msgpoll" on page 208)  
| | • For IP/US statistics reporting: lapi_statistics_t (see "LAPI_Qenv" on page 221)  
| | • For UDP support:  
| | – lapi_add_udp_port_t (see "LAPI_Util" on page 241)  
| | – lapi_extend_t, lapi_udp_t, lapi_udpinfo_t (see "LAPI_Init" on page 199)  
| | • For utility functions and DGSM data transfers (see "LAPI_Util" on page 241):  
| | – ddm_func_t  
| | – lapi_add_udp_port_t  
| | – lapi_amdgsp_t  
| | – lapi_dg_handle_t  
| | – lapi_dgsm_block_t  
| | – lapi_dgsm_control_t  
| | – lapi_dgsm_copy_t  
| | – lapi_dgsm_gosub_t  
| | – lapi_dgsm_iterate_t  
| | – lapi_dgsm_mcopy_t  
| | – lapi_dgsp_descr_t  
| | – lapi_dref dgsp_t  
| | – lapi_pack_dgsp_t  
| | – lapi_reg_ddm_t  
| | – lapi_reg dgsp_t  
| | – lapi_resv_dgsp_t  
| | – lapi_unpack_dgsp_t  
| | – lapi_usr_fcall_t  
| | – lapi_util_t |

Tips for LAPI users

To take advantage of some of LAPI’s latest features, you may need to make changes to your existing LAPI programs. For example — for the inline completion handler, the header handler needs to be modified to return more information to LAPI. See "The enhanced header handler interface" on page 95.

The current versions of LAPI maintain source and binary compatibility with previous versions of LAPI. Existing LAPI binaries will run under IBM’s TWS LoadLeveler® and Parallel Environment (PE) licensed programs.

Your existing LAPI programs can also be run standalone. The term standalone refers to a Linux system running LAPI over User Datagram Protocol / Internet Protocol (UDP/IP) or an RSCT LAPI for AIX system that is not running PE. See Chapter 17, “Using LAPI on a standalone system,” on page 141.

For noncontiguous data transfers that are not easily described by the vector datatypes, data gather/scatter programs (DGSPs) may provide an alternative way to move data from one address space to another. See “Using data gather/scatter programs (DGSPs)” on page 52.
LAPI for Linux and RSCT LAPI for AIX include an updated set of error codes. See "LAPI error codes" on page 309.

LAPI for Linux and RSCT LAPI for AIX include many of the same features and functions. Here are the major differences:

<table>
<thead>
<tr>
<th>Table 13. Differences among LAPI versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSCT LAPI for AIX 6.1:</td>
</tr>
<tr>
<td>Is part of AIX 6.1</td>
</tr>
<tr>
<td>Supports the High Performance Switch (HPS) and certain models of the InfiniBand switch</td>
</tr>
<tr>
<td>Provides support for bulk transfer using RDMA operations</td>
</tr>
<tr>
<td>Provides support for user-initiated data transfer using RDMA operations on the HPS</td>
</tr>
<tr>
<td>Provides support for checkpoint and restart operations</td>
</tr>
<tr>
<td>Provides barrier synchronization register (BSR) support</td>
</tr>
<tr>
<td>Includes the shared memory kernel extension</td>
</tr>
<tr>
<td>Can be run standalone (except on InfiniBand systems running over US)</td>
</tr>
</tbody>
</table>

Migration and coexistence considerations

LAPI version 2.4.7 (for AIX 5.3) and LAPI version 3.1.4 (for AIX 6.1) are only compatible with and supported on PE 5.2 and TWS LoadLeveler 4.1. These versions of LAPI are not supported on and should not be installed with earlier versions of PE and TWS LoadLeveler. Current customers running PE and TWS LoadLeveler might need to uninstall earlier versions of PE and TWS LoadLeveler in order to install the latest version of LAPI.

There is no coexistence among or migration path for prior versions of PE, LAPI, and TWS LoadLeveler. It is recommended that LAPI 2.4.7 (for AIX 5.3) or LAPI 3.1.4 (for AIX 6.1), PE 5.2, and TWS LoadLeveler 4.1 be installed at the same time.

Parallel application programs that use LAPI must use the identical level of software.

For information about binary compatibility issues for 32-bit applications that use data striping on AIX, see "Communication and memory considerations for AIX" on page 133.

Descriptions and formats of the MP_COMMON_TASKS, MP_LAPI_INET_ADDR, MP_LAPI_NETWORK environment variables are provided in this book for informational purposes only. These environment variables are not intended to be used as external programming interfaces. IBM will not guarantee that the formats or values of these variables can continue to be used without change in future releases. Programmers and users who choose to develop applications that depend on these variables do so with the understanding that these variables may be subject to future change. IBM cannot guarantee that such applications can migrate.
or coexist with future releases without additional changes, nor will IBM ensure that there will be binary compatibility of these variables.
Chapter 2. What is the low-level application programming interface (LAPI)?

The low-level application programming interface (LAPI) is a message-passing API that provides a one-sided communication model. In this model, one task initiates a communication operation to a second task. The completion of the communication does not require the second task to take a complementary action.

The LAPI library provides basic operations to "put" data to and "get" data from one or more virtual addresses of a remote task. LAPI also provides an active message infrastructure. With active messaging, programmers can install a set of handlers that are called and run in the address space of a target task on behalf of the task originating the active message. Among their other uses, these handlers can be used to dynamically determine the target address (or addresses) where data from the originating task must be stored. You can use this generic interface to customize LAPI functions for your environment.

Some of LAPI’s other general characteristics include:

- Flow control
- Support for large messages
- Support for generic non-contiguous messages
- Non-blocking calls
- Interrupt and polling modes
- Efficient exploitation of switch functions
- Event monitoring support (to simulate blocking calls, for example) for various types of completion events

LAPI is meant to be used by programming libraries, and by power programmers for whom performance is more important than code portability.

To use LAPI, you need to understand the following basic concepts and the related LAPI characteristics:

- **LAPI handle**
  You interact with LAPI through an opaque object called a LAPI handle. This object is also referred to as a LAPI instance or a LAPI context. Almost without exception, LAPI function calls take a LAPI handle as the first argument.

- **Data buffer**
  You provide data to LAPI for reading and writing. For contiguous data transfer, a data buffer is defined by a base address and data length. LAPI also provides a number of methods for transferring non-contiguous data, such as multiple buffers, repeating block/stride descriptions, and data gather/scatter programs (DGSPs).

- **Origin and target**
  LAPI communication operations usually involve two tasks. For these operations, origin (or source) denotes the task that initiates the LAPI operation and target (or destination) denotes the task where the address space is accessed during the operation. The origin and target can be the same for any of the LAPI communication calls, but if the origin and target data areas overlap, the result of the communication is undefined.
• **Push and pull operations**
  A *push* operation transfers data from the origin task to the address space of the target task. A *pull* operation transfers data from the address space of a target task into the (local) address space of the origin task.

• **Blocking and non-blocking calls**
  A *blocking* procedure returns only after the operation is complete. All LAPI synchronization calls are blocking. There are no restrictions on the modification of user resources.

  A *non-blocking* procedure might return before the operation is complete and before you can modify all of the resources that are specified in the call. All LAPI data transfer calls are non-blocking. A non-blocking operation is considered to be complete only after a function or an event that tests for completion indicates that the operation is complete. LAPI provides counters and handlers to signal completion of various events for non-blocking calls.

  Completion of LAPI communication operations can be detected either by checking the values of counters associated with LAPI operations or by the completed execution of user-specified handlers associated with these operations. To obtain the semantics of blocking communication with LAPI, you can combine a LAPI communication operation with procedures that wait on LAPI completion events.

• **Counters and handlers**
  LAPI uses counters and handlers to notify you about such events as the arrival of a message or the completion of a message. A *counter* is an opaque object; only its value is of interest. A *handler* is a callback routine that you provide. LAPI updates a counter, calls a handler, or both to notify you about an event. In terms of notification latency, handlers are generally more efficient than counters. When a handler is called, your program takes control immediately. On the other hand, your program has to poll on a counter to know about any updates.

  With a few notable exceptions, the use of counters and handlers is optional in LAPI communication calls. For any counters you specify, LAPI increments the counter at certain points in the message delivery sequence. Similarly, LAPI invokes any optional callback handlers you specify at the appropriate point in the operation.

• **Completion of communication operation**
  A communication operation is considered to be *complete*, (with respect to the buffer) when the buffer is reusable.

  With respect to the *origin buffer*:
  – a push operation is complete when the data has been copied out of the buffer at the origin task and can be overwritten
  – a pull operation is complete when the origin buffer holds the new data that was obtained by the pull operation

  With respect to the *target buffer*:
  – a push operation is complete when the new data is available at the target buffer
  – a pull operation is complete when the data has been copied out of the target buffer and the target task can overwrite that buffer

  Two types of communication behavior support two different definitions of *completion*:
  1. In *standard* behavior, a communication operation is complete:
     – at the origin task when it is complete with respect to the origin buffer
     – at the target task when it is complete with respect to the target buffer
2. In *synchronous* behavior, a communication operation is complete:
   - at the origin task when it is complete with respect to both the origin buffer
     and the target buffer
   - at the target task when it is complete with respect to the target buffer

Both standard and synchronous behaviors can be obtained for LAPI push
operations; however, only synchronous behavior can be obtained for LAPI pull
operations. When using send completion handlers for notification of message
completion, it is important to note that this only applies to the standard behavior
as defined above, for push operations.

**Message ordering and atomicity**

Two LAPI operations that have the same origin task are considered to be
ordered *with respect to the origin* if one of the operations starts after the other
operation has completed at the origin task. Similarly, two LAPI operations that
have the same target task are considered to be ordered *with respect to the target*
if one of the operations starts after the other operation has completed at the
target task. If two operations are not ordered, they are considered *concurrent.*

LAPI provides no guarantees of ordering for concurrent communication
operations. However, LAPI does provide mechanisms that an application can use
to guarantee order.

As an example, consider the case where a node issues two standard behavior
push operations to the same target node, where the target buffer regions overlap.
These two operations may complete in any order, including the possibility of the
first push operation overlapping in time with the second push operation. The
contents of the overlapping region will be undefined, even after both push
operations complete. Using synchronous behavior for both push operations
(waiting for the first to complete before starting the second) will ensure that the
overlapping region contains the result of the second after both push operations
have completed.

**Error handling**

LAPI provides you with the option of registering an error handler during LAPI
initialization. LAPI calls the error handler with an error code that is passed as a
parameter to the handler when it encounters a fatal error that would normally
cause LAPI to terminate. If no error handler is registered, LAPI terminates the job
when such fatal errors occur. LAPI also provides functions to translate a LAPI
error code — an integer value — that is passed in to the registered error handler
into a more explanatory message string.

If an error occurs during a communication operation, the error may be signaled at
the origin of the operation, the target of the operation, or both. Some errors may
be caught before the communication operation begins; these are signaled at the
origin. However, some errors will not occur until the communication is in progress
(a segmentation violation at the target, for example); these may be signaled at
either end (or at both ends) of the communication.

**Progress**

Most LAPI communication calls are non-blocking and control may thus be
returned to you without the communication completing. Other LAPI calls are
therefore needed to make progress with these pending communications and to
drive them to completion. Various LAPI subroutines drive the progress of LAPI
communication explicitly or implicitly by invoking a communication *dispatcher* that
is internal to LAPI.

**Polling mode and interrupt mode**

You can run LAPI in either polling mode or interrupt mode. In *polling mode*, the
sending and receiving of messages only happens when you explicitly call a LAPI
function. In *interrupt mode*, a receive interrupt is generated for incoming
messages when your program is not in any LAPI function call. An extra thread, which LAPI creates at initialization, is called to handle the interrupt.

- **Statistics collection**
  Using LAPI’s query functions, you can query statistics related to data that is transferred using the user space (US) protocol or the User Datagram Protocol / Internet Protocol (UDP/IP), through intra-task local copy or shared memory. In addition, you can print data transfer statistics.
  
  For more information, see Chapter 9, “Collecting statistics and querying resources,” on page 81.

- **Profiling**
  LAPI’s profiling interface includes wrappers for each LAPI function, so you can collect data about each of the LAPI calls. For example, you can write a program that records the message size that is used in each call.
  
  For more information, see Chapter 10, “Using LAPI’s profiling interface,” on page 85.

- **Lock sharing**
  Sharing locks with LAPI provides increased efficiency in protocol layering and user programming. When you need to use a locking mechanism to protect your programs’ data structures, you can use the same locking mechanism that is employed by LAPI through its lock sharing interface. This way, your program is more tightly coupled with LAPI in terms of locking. Using a shared lock in your program may result in improved latency and throughput, as compared to using a separate lock.
  
  For more information, see Chapter 14, “Lock sharing,” on page 111.

- **Failover and recovery**
  Using Tivoli® Workload Scheduler (TWS) LoadLeveler command file settings or POE environment variables, you can request that LAPI use more than one adapter for each of the job tasks. For jobs that are run using this configuration, if one of the adapters allocated to the tasks of a job fails during the course of the job run, the job will continue and LAPI will use the available adapters for communication. If the failed adapter recovers the ability to communicate while the job is running, LAPI recovers the use of this failed adapter for future communication during the remainder of the job run.
  
  For system requirements and additional information, see: Chapter 15, “Striping, failover, and recovery,” on page 127.

- **Striping**
  LAPI can manage the distribution of bulk transfer data among the various communication adapters that are assigned for communication, thereby providing LAPI clients with improved performance with regard to communication bandwidth. By using striping in conjunction with the bulk transfer transport mechanism, LAPI clients can experience gains in communication performance that scale linearly with the number of adapters (up to a limit of eight) for sufficiently-large messages. Messages that do not use the bulk transfer communication mode cannot benefit practically from striping over multiple adapters, so LAPI uses only one of the assigned adapters for communication when bulk transfer is turned off with environment variable settings or for small messages, which do not use bulk transfer. Even when striping is not done, the other assigned adapters serve as backups for LAPI’s failover and recovery function.
  
  For system requirements and additional information, see: “Data striping” on page 132.

- **User-initiated RDMA transfer**
To improve bandwidth for large messages and offload communication work from the CPUs to the adapters, LAPI provides a generic transfer function that supports remote direct memory access (RDMA) on AIX using the HPS. With this function, data can be transferred to and from a remote location without any CPU involvement on the remote side.

For system requirements and additional information, see: “User-initiated RDMA transfer on AIX” on page 65.

• **Standalone operation**
  
  You can use LAPI with or without the parallel operating environment (POE) component of the IBM Parallel Environment (PE) licensed program. When you use it without PE, LAPI is referred to as operating in *standalone mode*. In certain situations, such as failure recovery, you will have greater flexibility if you use LAPI in standalone mode, as opposed to the enhanced usability that comes when you use it with PE.

  For system requirements and additional information, see: Chapter 17, “Using LAPI on a standalone system,” on page 141.

**Why use LAPI?**

LAPI provides the following advantages over other messaging layers:

• **Performance**
  
  LAPI provides basic functions for optimal performance. In particular, LAPI provides low latency for short messages using the user space (US) protocol and high bandwidth for large messages.

• **Flexibility**
  
  Compared to other communication protocols, such as the Message Passing Interface (MPI) and the Internet Protocol (IP), LAPI provides a lower-level interface to the switch.

  With LAPI’s messaging support over UDP/IP, applications that are written using LAPI can be run over any cluster of processors running AIX or Linux.

  LAPI’s one-sided communication model provides flexibility because the completion of an operation by one task does not require any other task to take a complementary action. This model is supported by the use of *virtual addressing*. Addresses in the target task address space are passed to LAPI communication calls by the origin task. These addresses can refer to remote data buffers, handlers, or counters. The use of target addresses allows data to be delivered without any need for explicit action by the target task.

  Two-sided communication can be simulated using LAPI’s communication calls in conjunction with various event-monitoring routines (such as waiting for counter values or execution of specified handlers, for example).

• **Reliability**
  
  Using LAPI guarantees delivery of messages. Errors that are not directly related to the application are not propagated back to the application.

• **Availability**
  
  On systems with multiple adapters per node, if an adapter fails while a job with settings that request the use of multiple adapters within the job tasks is running, LAPI switches communication over to another, available adapter. If the original adapter regains the use of the connection during the course of the job run, LAPI recovers the use of this adapter for communication for the remainder of the job run. On AIX systems, checkpoint and restart operations continue to be supported during these failures and recoveries.

• **Extendibility**
LAPI supports programmer-defined handlers that are called when a message arrives, so you can customize LAPI for your specific environment. For example, you can write completion handler functions to perform communication that helps maintain user program state. With such handlers, you can, in effect, extend LAPI's active message functionality to do your specific state update operations for you.
Chapter 3. An overview of LAPI

This chapter provides a functional overview of the various subroutines that constitute the low-level application programming interface. LAPI subroutines provide a wide variety of functions that can be used efficiently and flexibly to obtain most behaviors required from any parallel programming API.

In general, LAPI functions:

- Are non-blocking calls.
- Provide polling mode and interrupt mode.
- Indicate the completion of a message or operation either by incrementing counters or by running user-specified handlers. Counters and handlers are available at both the sending and receiving side, depending on the API call.
- Provide C and FORTRAN subroutine bindings.
- Provide `extern "C"` declarations for C++ programming.
- Provide profiling interfaces for C, C++, and FORTRAN programs.
- Do not guarantee order of message delivery.

Complementary functions provide for checking completion of operations and for enforcing relative ordering if required. Additionally, LAPI functions allow tasks to exchange addresses that will be used in LAPI operations.

LAPI functions (and the related subroutines) include:

- Functions to initialize and terminate LAPI: `LAPI_Init`, `LAPI_Term`.
- Functions to query and set up the runtime environment: `LAPI_Qenv`, `LAPI_Senv`.
- Address-related functions: `LAPI_Address`, `LAPI_Address_init`, `LAPI_Address_init64`, `LAPI_Addr_get`, `LAPI_Addr_set`. The `LAPI_Address_init64` subroutine treats all data as 64-bit values, so it can support communication between 32-bit and 64-bit tasks.
- Put and get functions: `LAPI_Put`, `LAPI_Get`, `LAPI_Xfer` with transfer types `LAPI_GET_XFER (GET)` and `LAPI_PUT_XFER (PUT)`.
- Active message functions: `LAPI_Amsend`, `LAPI_Xfer` with transfer type `LAPI_AM_XFER (AM)`.
- Non-contiguous data transfer functions: `LAPI_Amsendv`, `LAPI_Getv`, `LAPI_Putv`, `LAPI_Xfer` with transfer types `LAPI_AMV_XFER (AMV)`, `LAPI_GETV_XFER (GETV)`, and `LAPIPUTV_XFER (PUTV)`.
- Remote read-modify-write functions: `LAPI_Rmw`, `LAPI_Rmw64`, and `LAPI_Xfer` with transfer type `LAPI_RMW_XFER (RMW)`. The `LAPI_Rmw64` subroutine treats all data as 64-bit values, so it can support communication between 32-bit and 64-bit tasks.
- A wrapper function for all generic data transfer calls that includes support for 32-bit/64-bit interoperability: `LAPI_Xfer`. `LAPI_Xfer` also provides an optional send completion handler for most calls and for "get" transfers, a receive completion handler.
- Functions for user-initiated RDMA transfer on AIX (using the HPS): `LAPI_Xfer` with transfer type `LAPI_RDMA_XFER (RDMA)`.
- Multicasting functions: `LAPI_Group_create`, `LAPI_Group_free` and `LAPI_Xfer` with transfer type `LAPI_MC_XFER (MC)`. 
A function to support low-latency API extensions: **LAPI_Xfer** with transfer type **LAPI_AM_LW_XFER (AM_LW)**.

A function for extended user header support: **LAPI_Xfer** with transfer type **LAPI_AMX_XFER (AMX)**.

An additional wrapper function that provides a variety of utilities: **LAPI_Util**.

Progress-monitoring functions: **LAPI_Getcntr**, **LAPI_Msgpoll**, **LAPI_Probe**, **LAPI_Setcntr**, **LAPI_Waitcntr**.

Message ordering functions: **LAPI_Fence**, **LAPI_Gfence**.

An error message function: **LAPI_Msg_string**.

Recovery-related functions for standalone systems (without PE, running LAPI for Linux over UDP/IP or RSCT LAPI for AIX): **LAPI_Nopoll_wait**, **LAPI_Purge_totask**, **LAPI_Resume_totask**, **LAPI_Setcntr_wstatus**.

### Initialization and termination

LAPI uses a number of internal structures to enable it to perform message-passing operations on behalf of the user. Memory must be allocated for these internal structures, and the structures must be appropriately initialized before any LAPI communication is performed. Correspondingly, when all LAPI communication is done, memory used by LAPI structures must be freed and potentially reused by the user program.

The **LAPI_Init** subroutine is used to allocate memory for LAPI's communication structures and to initialize them. It returns a unique handle that represents a single LAPI communication context. This handle is subsequently passed as a parameter to each of the other LAPI functions. **LAPI_Init** takes in a parameter of type **lapi_info_t**. The fields in this structure are used to specify various initialization parameters. **LAPI_Init** reads in various environment variables and sets up various communication channels based on the values of these variables. For example, the user can set environment variables to indicate whether communication will take place using the user space (US) protocol or the user datagram protocol (UDP) and whether to use shared memory.

The **LAPI_Term** subroutine is used to free memory associated with LAPI's communication structures. It takes a LAPI handle as a parameter and uses it to terminate the corresponding communication context. Once **LAPI_Term** is called, no further LAPI communication can be performed on the handle that has been terminated. Typically, **LAPI_Init** is called once at the beginning of the user program and **LAPI_Term** is called just before the user program terminates. However, LAPI allows a handle to be initialized after it has been terminated.

### Querying and setting up the runtime environment

A number of variables constitute LAPI's runtime state. Many of these variables can be queried at runtime. For example, it is often useful (if not always required) to know the number of tasks in a given job as well as the identity of the current task and to design the user program to take actions according to their values. Many of LAPI's runtime state variables can also be set to alter LAPI's behavior through the job execution and to tune LAPI's performance. For example, it may be useful to turn off interrupts to signal incoming packets when the user program explicitly makes a number of calls to various LAPI progress routines.
The **LAPI_Qenv** subroutine is used to query elements of LAPI’s runtime state. The **LAPI_Senv** subroutine correspondingly allows the programmer to set the value of various elements of LAPI’s runtime state. LAPI defines an enumeration of query types (**lapi_query_t**). A parameter of this type is passed to **LAPI_Qenv** or **LAPI_Senv** to indicate the value to query or set, respectively.

**Address-related functions**

Many of LAPI’s communication operations take advantage of virtual addresses in the remote task’s address space. An address might refer to a buffer location or a handler function on the target task. For example, LAPI’s active message functions are passed the address of a header handler that executes on the target task upon the arrival of the first packet of a message.

The **LAPI_Address_init** collective operation allows tasks to exchange virtual addresses of mutual interest. Such a function is especially required in the typical scenario where tasks do not have identical address maps. The **LAPI_Address_init** function takes in a virtual address and the address of a local buffer that is large enough to hold one address from each of the tasks in the job. When the function returns, the local buffer has virtual addresses from each of the tasks, at indices that correspond to each task’s ID.

The **LAPI_Address_init64** collective operation is similar to **LAPI_Address_init**, but handles 64-bit operand addresses and 32-bit operand addresses.

LAPI also provides operations to register and retrieve active message header handlers as specific indices (referred to as *address handles*) in a table of handler functions that is maintained by LAPI. These user-defined address handles can be used instead of the virtual addresses of the header handler functions. The **LAPI.Addr_set** subroutine registers a header handler function at the index passed in by the caller. The **LAPI.Addr_get** subroutine retrieves the address of the header handler function stored at a given index. If all tasks of the job are programmed to register and use the same indices for specific header handler functions, a collective **LAPI_Address_init** call to set up the address table can be avoided.

**FORTRAN programmers:** Because there is no concept of address (&) in FORTRAN, LAPI provides the **LAPI_Address** subroutine, which FORTRAN programs can call when a value needs to be treated as an address.

**Put and get functions**

LAPI includes subroutines that provide the user with a Remote Memory Copy interface, allowing the direct transfer of contiguous data to or from the virtual address of a remote task. These subroutines support “pull” and “push” operations. The **LAPI_Get** subroutine copies (or “pulls”) data from the address space of a target task into the address space of the origin task. The **LAPI_Put** subroutine copies (or “pushes”) data into the address space of a target task from the address space of the origin task. Other primary characteristics of the **LAPI_Put** and **LAPI_Get** functions include the following:

- The basic data transfer operations are memory-to-memory copy operations that transfer data from one virtual address space to another virtual address space.
The operations are unilateral. One task initiates an operation, but the completion of the operation does not require the other task to take some complementary action. (This model is different from the more common send-receive communication model, in which a send operation from one task requires a complementary receive operation with matching parameters at the target task to be posted for completion.)

The initiating task specifies the virtual address of the source of the data and the virtual address of the destination of the data (as opposed to a send and receive task, in which each side specifies the address in its own address space). As described in "Address-related functions" on page 19, LAPI provides the `LAPI_Address_init` and `LAPI_Address_init64` subroutines to obtain virtual addresses in a remote task’s address space to or from which data needs to be copied. These virtual addresses must be obtained before making the `LAPI_Put` and `LAPI_Get` calls.

Because data transfer operations are unilateral and no synchronization between the two tasks is implied, additional primitives are provided for explicit task synchronization when such synchronization is necessary for program correctness.

### Active messages

In many cases, it is impractical to exchange virtual addresses for all communication performed during a job. Instead, it is useful to decide on the buffer address to which incoming data must be stored when the data arrives at the target, based on information about the state of the target task. Such a decision would be taken, for example, by executing a function in the target task’s address space that examines data in the incoming message and the state of the target task to decide where the data must be stored. It is also similarly useful to be able to define and execute a user-specified procedure at the target task as soon as a message is completely transferred into the target task’s address space. Such a procedure can be used, for example, to automatically process the message data at the target and to send any necessary reply messages.

LAPI’s active message subroutine, `LAPI_Amsend`, provides users with this ability to calculate the target buffer address at the destination by allowing the caller to specify the address of a header handler function in the target task’s address space as one of its parameters. The header handler function is executed at the target as soon as the first packet of a message arrives and returns the address of the target task buffer where the message data must be stored. The header handler function may also change reference parameters passed into it to specify a completion handler function and a parameter that should be passed to it. The completion handler function, if specified, will be executed by LAPI as soon as the entire message is transferred into the address returned by the header handler. The `LAPI_Amsend` subroutine transfers contiguous messages only.

LAPI’s underlying *active message* infrastructure has the following characteristics:

- It includes the address of a user-specified handler. When the active message arrives at the target task, the specified handler is called. This handler runs in the address space of the target task.
- Optionally, the active message might also bring with it user header data from the originating task that could aid in the computation of the target buffer address by the header handler function, or in the future processing of actual message data.
- Operations are unilateral in the sense that the LAPI client on the target task does not have to take explicit action for the active message to complete.
• The handler that is called must return the address of storage buffers for arriving data.

• If the \texttt{LAPI\_Addr\_set} subroutine (described in “Address-related functions” on page 19) is used to register the address of a header handler at a specific address handle, that address handle may be passed in to \texttt{LAPI\_Amsend} in place of the header handler address.

### Non-contiguous data transfer

The \texttt{LAPI\_Put} and \texttt{LAPI\_Get} subroutines (described in “Put and get functions” on page 19) and the \texttt{LAPI\_Amsend} subroutine (described in “Active messages” on page 20) can be used to transfer data that is laid out in contiguous portions of memory. Very often, though, data that is laid out in non-contiguous memory locations may need to be transferred between tasks. Such non-contiguous data is often referred to as a vector. Although it is possible to transfer a non-contiguous vector through multiple \texttt{LAPI\_Put}, \texttt{LAPI\_Get}, or \texttt{LAPI\_Amsend} calls, it is much more efficient to simply specify the layout of the data at the source or target task and make a single LAPI call that transfers data between the specified non-contiguous regions. At a minimum, having such a call reduces the locking overhead associated with every LAPI call that is incurred to maintain the integrity of LAPI’s internal data structures.

LAPI provides a \texttt{lapi\_vec\_t} datatype that can be used to describe the layout of non-contiguous (vector) data and three subroutines (\texttt{LAPI\_Putv}, \texttt{LAPI\_Getv}, and \texttt{LAPI\_Amsendv}) to transfer this data in a similar manner to \texttt{LAPI\_Put}, \texttt{LAPI\_Get}, and \texttt{LAPI\_Amsend}. \texttt{LAPI\_Putv} transfers vector data from the calling task to a target task, \texttt{LAPI\_Getv} transfers vector data from a target task to the caller task, and \texttt{LAPI\_Amsendv} specifies a header handler function that is executed at the message target to obtain a vector description of where data must be stored at the target. Descriptions of the source and target vectors, as pointers to the \texttt{lapi\_vec\_t} types, are passed in to the \texttt{LAPI\_Putv} and \texttt{LAPI\_Getv} subroutines. The \texttt{LAPI\_Amsendv} subroutine only requires the source vector description. A target vector description is returned by the header handler in a manner similar to the way a single buffer is returned by the header handler in a \texttt{LAPI\_Amsend} call. The vector description with the \texttt{lapi\_vec\_t} structure includes information that directly or indirectly indicates the starting buffer addresses and lengths of each of the non-contiguous source and target regions. LAPI does not require that non-contiguous data at the source and destination be laid out in identical fashion.

### Remote read-modify-write functions

Multi-threaded, single-process programs often use atomic read-modify-write (RMW) operations to check and change the values of shared variables from multiple threads, thereby avoiding possible inconsistencies due to the effects of caching in the underlying architecture. To avoid more significant locking overheads, RMW operations are often used to synchronize among threads. With the \texttt{LAPI\_Rmw} subroutine, a calling task in a parallel, multi-process program can atomically check and change the value of a remote task’s variable. For example, you can use \texttt{LAPI\_Rmw} to synchronize a pair of tasks in a job based on the value of a “shared” variable that is located on one of the tasks. \texttt{LAPI\_Rmw} provides you with an alternative to using other LAPI calls that facilitate synchronization among all of the tasks in a job.

With \texttt{LAPI\_Rmw}, the operation is performed at the target task on the remote address that is specified by the target variable. The operation takes in one or more input values from the origin and performs one of four selected operations on a
variable from the target task using these origin task input values. Depending on the result of this operation, LAPI replaces the target task variable with a value dictated by the result of the operation. The previous value of the target variable is then returned to the origin by modifying an origin task variable. This variable's address is passed in as one of the parameters to the LAPI_Rmw subroutine. There are four read-modify-write operations:

1. **SWAP** replaces the target task variable with the value specified by the calling task and returns the original value of the target task variable by setting the specified origin task variable.
2. **COMPARE_AND_SWAP** compares an input value with the value of the target variable, and if they are equal, replaces the target task variable with another specified input value.
3. **FETCH_AND_ADD** adds the specified input value to the target task variable and stores the result in the target task variable. **FETCH_AND_ADD** returns the original value of the target task variable (before performing the addition) by setting the specified origin task variable.
4. **FETCH_AND_OR** does a bit-wise OR of the specified input value with the value in the target task variable and stores the result in the target task variable. Like **FETCH_AND_ADD**, **FETCH_AND_OR** also returns the original value of the target task variable.

Completion is signaled at the origin if an origin counter is specified with the operation. It is important to note that LAPI_Rmw operations are atomic with respect to the target, but not necessarily with respect to the origin. That is, two successive LAPI_Rmw calls from a task to the same origin variable at a target task may be executed in any order at the target task, with the only guarantee that each will execute atomically at the target.

The LAPI_Rmw64 subroutine performs the same function as LAPI_Rmw for a 64-bit data structure.

### Generic data transfer

The contiguous and non-contiguous LAPI subroutines described in the previous sections have some limitations. These limitations include: (1) there is no option to specify a send completion handler that can be executed when the source buffer is reusable, so the sending side must use counters to determine origin buffer availability; (2) there is no support for 32-bit and 64-bit interoperability, and (3) there is no support for flags that can be used to direct transfer-specific behavior.

These limitations are resolved by the LAPI_Xfer subroutine. This subroutine takes a LAPI handle and a pointer to a union of structures as arguments. Each structure in the union corresponds to one of the data transfer functions described in the previous sections, with the first element of the structure specifying the type of transfer, such as "put", "get", or "active message". Each structure contains fields that correspond to the parameters in the corresponding LAPI data transfer call (for example, the structure for the "put" transfer type has fields for each of the parameters in LAPI_Put), with type modifications and additional fields to achieve the enhanced functionality mentioned above. For example, each structure has a field for an optional send completion handler (or, in the case of "get", for a receive completion handler), and a flags field to control specific behavior relating to the data transfer. In addition, all address parameters that belong to a remote task (such as the target counter address) are specified as a new lapi_long_t type, which allows remote 64-bit addresses to be represented on 32-bit tasks. There is a LAPI_Xfer call corresponding to each of the LAPI data transfer calls described in the previous
sections, with the **LAPI_Xfer** call supporting a functional superset of the corresponding LAPI data transfer call. Data transfer between 32-bit and 64-bit tasks must use **LAPI_Xfer**.

The **LAPI_Xfer** subroutine provides a transfer type for non-contiguous messages that has no corresponding LAPI data transfer call. This **LAPI_Xfer** call is somewhat analogous to the **LAPI_Amsendv** call with one major difference: the layout of non-contiguous data is described for this transfer type using commands of a simple, assembly-type language. Such a description of the layout of non-contiguous data is called a **data gather/scatter program (DGSP)**. A DGSP description of non-contiguous data can be used to capture more powerful datatypes than can be captured by LAPI's vector description structures alone (for example, recursive datatypes can be more easily described with a DGSP). Before it is used, a DGSP must be registered with the **LAPI_Util** subroutine. **LAPI_Util** returns a DGSP handle that can be used to refer to the corresponding DGSP. See “**LAPI_Util**” on page 241 for more information.

The **LAPI_Xfer** call to transfer non-contiguous data using a DGSP takes in a DGSP handle describing the source data layout as input and transfers data to the destination by running the corresponding DGSP. At the destination, the data can be written either contiguously or written in a manner described by a different DGSP. A DGSP handle describing the data layout at the target must be returned through one of the parameters of an enhanced (but binary-compatible) header handler interface (see “**The header handler**” on page 57 for more information). This DGSP is then run at the target to store the non-contiguous message data into appropriate target buffers. For more information, see “**LAPI_Xfer**” on page 260.

### Extended user header support

To support extended user headers, LAPI includes a generic transfer function — **LAPI_Xfer** with transfer type **LAPI_AMX_XFER (AMX)**. **LAPI_AMX_XFER** provides a superset of the **LAPI_AM_XFER** and **LAPI_DGSP_XFER** functions. With these two transfer functions, you can specify user headers that are up to one packet in length. With **AMX**, you can specify user headers that are up to 2 GB in length.

When using a DGSP to transfer data, you can use **LAPI_AMX_XFER** to set the starting offset of the DGSP, as opposed to using **LAPI_DGSP_XFER**, which always starts at 0. In addition, you can specify a starting offset for the receive-side DGSP using the **recv_offset_dgsp_bytes** field, which is in the **lapi_return_info_t** structure that is returned from the header handler.

### Multicasting

To support multicasting, LAPI includes two new subroutines (**LAPI_Group_create** and **LAPI_Group_free**) and a generic transfer function — **LAPI_Xfer** with transfer type **LAPI_MC_XFER (MC)**.

### Low-latency API extensions

To optimize communication for very small messages, LAPI supports a generic transfer function: **LAPI_Xfer** with transfer type **LAPI_AM_LW_XFER (AM_LW)**. This function, which provides the the lowest latency (and thus, the fastest communication), can be used for messages that are a maximum of 128 bytes. This size includes the user data and the user header (if any). One of this function's restrictions is that it does not support counters. This and other restrictions allow for...
User-initiated RDMA transfer

To improve bandwidth for large messages and offload communication work from the CPUs to the adapters, LAPI supports remote direct memory access (RDMA) on AIX (using the HPS) with a generic transfer function: \texttt{LAPI\_Xfer} with transfer type \texttt{LAPI\_RDMA\_XFER (RDMA)}. With \texttt{RDMA}, data can be transferred to and from a remote location without any CPU involvement on the remote side. Two types of RDMA operations are supported: \texttt{LAPI\_RDMA\_GET} and \texttt{LAPI\_RDMA\_PUT}. With \texttt{LAPI\_RDMA\_GET}, data from a remote task is transferred to a local task. With \texttt{LAPI\_RDMA\_PUT}, data is transferred to a remote task. For more information, see \textquote{\texttt{lapi\_hwxfer\_t details} on page 271}.

Using the RDMA transfer function requires resources known as \textit{RDMA contexts} to be acquired from the remote task and the registration of local and remote memory involved in the RDMA operation. To support these two operations, LAPI provides two new utility functions: \texttt{LAPI\_REMOTE\_RCXT} and \texttt{LAPI\_XLATE\_ADDRESS}. Once memory has been registered, it can be used in RDMA operations until it is explicitly deregistered. Similarly, remote RDMA contexts can be used in various RDMA operations until they are explicitly released. For more information, see \textquote{\texttt{LAPI\_Util} on page 241}.

Utility functions

The \texttt{LAPI\_Util} subroutine serves as a wrapper function for such data gather/scatter operations as registration and reservation, for updating UDP port information, for obtaining pointers to locking and signaling functions that are associated with a shared LAPI lock, and for setting up user-initiated RDMA operations on AIX.

Monitoring progress

As discussed in \textquote{Chapter 2}, progress operations are necessitated, when in polling mode, by the asynchronous nature of LAPI communication, and are provided by LAPI to drive pending communication operations to completion. These functions are used to explicitly invoke an internal LAPI communication dispatcher. The dispatcher will ensure that progress is made on any pending send or receive operations. If LAPI is running in interrupt mode, the progress operations are not strictly necessary.

The \texttt{LAPI\_Probe} subroutine is used in polling mode to explicitly invoke LAPI’s internal communication dispatcher once. This subroutine is typically called in a loop that also checks for the occurrence of some completion event (for example, an update to a LAPI counter value or an internal variable updated by the execution of a handler function).

A LAPI user may want to monitor a number of events, such as availability of the source buffer, availability of the target buffer, or completion of message transmission. LAPI provides the origin counter, target counter, and completion counter, respectively, to monitor each of these three events. LAPI also provides methods to set and monitor the values of counters. In addition, LAPI provides users with the ability to specify a send completion handler that is executed at the source when the source buffer is reusable and a (receive) completion handler that is executed at the target when message data has completely arrived at the target.
buffer. (As discussed in “Generic data transfer” on page 22, send completion handlers are available only with the LAPI Xfer calls.)

The following functions provide the means for a task to manage the completion state of LAPI operations using LAPI counters:

- **LAPI_Waitcntr** waits on a counter to reach a specified value and returns when the counter is equal to or greater than that value (blocking). This subroutine implicitly drives LAPI progress until the counter reaches the awaited value.

- **LAPI_Getcntr** gets the current value of a specified counter (non-blocking). This subroutine is typically used in conjunction with LAPI_Probe to drive progress while polling for counter values.

- **LAPI_Setcntr** sets the counter to a specified value.

- **LAPI_Nopoll_wait** waits for a counter update without polling (that is, without explicitly invoking LAPI’s internal communication dispatcher).

These subroutines also provide an efficient way to order the flow of LAPI operations or the use of such user-managed resources as buffers. For example, a series of LAPI_Put calls to a single target and buffer requires that the contents of the buffer at the target remains in step with the order of execution of the LAPI_Put calls at the origin. Using the cmpl_cntr counter in the LAPI_Put subroutine in conjunction with the LAPI_Waitcntr subroutine provides the necessary ordering. The use of (receive) completion and send completion handlers allows a client to rely exclusively on asynchronous notification rather than polling for message completion. The send completion handler does not guarantee that data has arrived at the target; however, it does guarantee that the user header buffers (if any) and data buffers can be reused. The receive completion handler is called at the target after the data transfer has completed.

The LAPI_Msgpoll subroutine is provided as a mechanism to probe continuously for send or receive completions. It polls for a fixed number of iterations or until some message has completed. The number of iterations for polling and the type (or types) of completion — send, receive, or both — that the subroutine must monitor are specified as parameters to the subroutine. LAPI_Msgpoll is especially useful when multiple messages are outstanding and the client is waiting for any one of them to complete.

### Message ordering

Because LAPI does not guarantee message order by default, it provides functions to enforce ordering among messages. The LAPI_Fence and LAPI_Gfence subroutines provide a way to enforce the order of execution of LAPI data transfer subroutines. LAPI subroutines that are initiated before these fencing operations are guaranteed to complete before LAPI functions initiated after the fencing functions. LAPI_Fence is a local operation used to guarantee that all LAPI operations initiated by the local task and the same task thread are complete. In contrast, LAPI_Gfence is a collective operation involving all tasks in the parallel program. A LAPI_Gfence operation combines local fencing on all tasks of the job with a barrier operation to synchronize the tasks in the parallel program. Both LAPI_Fence and LAPI_Gfence operations perform a data fence, only guaranteeing that data movement is complete. They do not perform an operation fence, which would need to include the completed execution of any pending active message completion handlers at the target.
Error message functions

The `LAPI_Msg_string` subroutine returns the message string that is associated with a LAPI return code.

Recovery-related functions

The following functions are recommended only for use on standalone systems (without PE, running LAPI for Linux over UDP/IP or RSCT LAPI for AIX):

- `LAPI_Nopoll_wait` waits for a counter update without polling (that is, without explicitly invoking LAPI's internal communication dispatcher).
- `LAPI_Purge_totask` allows a task to cancel messages to a given destination.
- `LAPI_Resume_totask` re-enables the sending of messages to the task.
- `LAPI_Setcntr_wstatus` sets an associated destination list array and destination status array to a counter. A corresponding `LAPI_Nopoll_wait` call accesses these arrays.
### Part 2. Basic LAPI tasks

#### Chapter 4. Installing RSCT LAPI for AIX
- Requirements ........................................ 29
  - Hardware ........................................ 29
  - Software ........................................ 29
    - For an AIX 6.1 environment .................. 29
    - For an AIX 5.3 environment .................. 30
  - How is RSCT LAPI for AIX packaged? ......... 31
  - RSCT LAPI for AIX filesets ................... 31
- Installation steps ............................... 32
- Post-installation steps ......................... 33
- Uninstallation steps ............................. 33

#### Chapter 5. Installing LAPI for Linux
- Requirements ....................................... 35
  - Hardware ........................................ 35
  - Software ........................................ 35
- How is LAPI for Linux packaged? ............ 36
- Installation steps ............................... 38
- Uninstallation steps ............................. 38

#### Chapter 6. Setting up, initializing, and terminating LAPI
- Setting and querying the LAPI environment ... 39
  - Setting environment variables .............. 39
    - LAPI communication modes ................ 41
  - Initializing LAPI ............................... 42
    - Passing information to LAPI using lapi_info_t ............ 42
    - Terminating LAPI ............................. 44

#### Chapter 7. Transferring data
- Data transfer operations ....................... 45
  - Flow of “put” operations .................... 45
  - Flow of “get” operations .................... 46
  - Flow of read-modify-write operations ...... 46
- Non-contiguous data transfer .................. 47
  - Using vectors ................................ 47
    - LAPI_GEN_GENERIC .......................... 49
    - LAPI_GEN_IOVECTOR ......................... 50
    - LAPI_GEN_STRIDED_XFER ..................... 51
    - Vector data transfer summary .............. 52
  - Using data gather/scatter programs (DGSPs) . 52
- Detecting completion ........................... 56
  - LAPI handlers ................................ 56
    - The header handler ........................ 57
    - The completion handler .................... 58
    - The send completion handler ............... 60
    - LAPI handler summary ....................... 61
- LAPI counters .................................... 61
  - Specifying target-side addresses .......... 62
  - Additional progress functions .............. 62
- Bulk message transfer on AIX .................. 63
- User-initiated RDMA transfer on AIX .......... 65
  - Resources required for an RDMA job ....... 65
  - RDMA setup .................................... 66
Chapter 4. Installing RSCT LAPI for AIX

This chapter includes information about installing the following versions of LAPI:

- version 3.1.4.0 of RSCT LAPI for AIX 6.1, which is part of the base AIX 6.1 operating system
- version 2.4.7.0 of RSCT LAPI for AIX 5L, which is part of the base AIX 5.3 operating system

For information about installing the Linux version of LAPI, see Chapter 5, "Installing LAPI for Linux," on page 35.

Requirements

To make use of all of RSCT LAPI's functions, you must make sure that the hardware and software requirements for your particular AIX environment are satisfied.

Hardware

In order to take advantage of all of the functions of RSCT LAPI for AIX, you must be using the following hardware:

- One of the following servers:
  - An IBM Power Systems server
  - An IBM BladeCenter Power Architecture server
- A minimum of two links per logical partition (LPAR).
- One of the following adapters:
  - For IP only:
    - 10/100 Ethernet adapter.
    - 1 GB Ethernet adapter.
    - 10 GB Ethernet adapter.
    - IP Over Fiber Channel adapter.
    - Virtual LAN adapter.
    - Virtual IP address support.
    - Etherchannel configuration support (an arrangement in which two or more network interfaces on a host computer are combined for redundancy or increased throughput).
  - For IP and US:
    - IBM High Performance Switch adapter.
    - GX Dual-Port 4x InfiniBand Host Channel Adapter (HCA).
    - Mellanox ConnectX dual port DDR IB 4X HCA PCIe 2.0 x8 adapter.
    - Topspin adapter over InfiniBand.
    - Shared Ethernet adapter.

Software

For an AIX 6.1 environment

To use all of the functions of RSCT LAPI for AIX 6.1, you must have the following software installed on your system:
AIX Version 6.1 (product number 5765-G62) with the 6100-04-01 Technology Level (or later).

In addition, you need these components if you are using the HPS:
- Version 1.2 (or later) of the switch network interface (SNI) component (the `devices.common.IBM.sni.rte` fileset).
- Version 1.3.0.1 (or later) of the system resource controller (SRC).
- Version 2.4.3.0 (or later) of the group services subcomponent of the Reliable Scalable Cluster Technology (RSCT) component of AIX 6.1. The `rsct.basic.rte` fileset includes group services.
- Version 3.1.4.0 (or later) of the LAPI subcomponent of RSCT — specifically, the `rsct.lapi.rte` fileset for LAPI, network resource table (NRT), and protocol network services daemon (PNSD) files.

- Version 5.2 (or later) of the Parallel Environment for AIX licensed program (product number 5765-PEA).
- Version 4.1 (or later) of the Tivoli Workload Scheduler (TWS) LoadLeveler licensed program (product number 5765-E69).
- One of the following compilers:
  - IBM C for AIX Version 6.0 (or later) (product number 5765-F57).
  - IBM XL C/C++ Enterprise Edition for AIX Version 7.0 (or later) (product number 5724-I11).
  - IBM XL FORTRAN for AIX Version 12 (or later) (product number 5724-I08).
  - VisualAge C++ Professional for AIX, Version 9.0 (or later) (product number 5765-F56).

For an AIX 5.3 environment
To use all of the functions of RSCT LAPI for AIX 5.3, you must have the following software installed on your system:
- AIX 5L Version 5.3 (product number 5765-G03) with the 5300-07-10 Technology Level (or later).

In addition, you need these components if you are using the HPS:
- Version 1.2 (or later) of the switch network interface (SNI) component (the `devices.common.IBM.sni.rte` fileset).
- Version 1.3.0.1 (or later) of the system resource controller (SRC).
- Version 2.4.3.0 (or later) of the group services subcomponent of the Reliable Scalable Cluster Technology (RSCT) component of AIX 5L 5.3. The `rsct.basic.rte` fileset includes group services.
- Version 2.4.7.0 (or later) of the LAPI subcomponent of RSCT — specifically, the `rsct.lapi.rte` fileset for LAPI, network resource table (NRT), and protocol network services daemon (PNSD) files.

- Version 5.2 (or later) of the Parallel Environment for AIX licensed program (product number 5765-PEA).
- Version 4.1 (or later) of the Tivoli Workload Scheduler (TWS) LoadLeveler licensed program (product number 5765-E69).
- One of the following compilers:
  - IBM C for AIX Version 6.0 (or later) (product number 5765-F57).
  - IBM XL C/C++ Enterprise Edition for AIX Version 7.0 (or later) (product number 5724-I11).
  - IBM XL FORTRAN for AIX Version 12.1 (or later) (product number 5724-I08).
  - VisualAge C++ Professional for AIX, Version 9.0 (or later) (product number 5765-F56).
**How is RSCT LAPI for AIX packaged?**

The RSCT LAPI for AIX filesets are packaged in a single `rsct.lapi` image. Most LAPI files are installed in the `/opt/rsct/lapi` directory. Exceptions are noted in "RSCT LAPI for AIX filesets." Links are created within the `/usr` tree to such commonly-needed files as `liblapi_r.a`, `lapi.h`, and the 32-bit version of `lapif.h`.

**RSCT LAPI for AIX filesets**

RSCT LAPI for AIX consists of the following filesets:

- **rsct.lapi.bsr**: Contains LAPI's barrier synchronization register (BSR) files.
  
  If you want to take advantage of the performance benefits of BSR, you must install this fileset. The same level of this fileset must be installed on all nodes in the cluster.

- **rsct.lapi.rte**: Contains RSCT LAPI for AIX's runtime environment, including headers and libraries. It is a prerequisite for the other LAPI filesets.

  The following files are installed from this fileset into `/opt/rsct/lapi`:

  - `include/lapi.h`: LAPI's C header file, with a link in `/usr/include`
  - `include/lapif.h`: LAPI's 32-bit FORTRAN header file, with a link in `/usr/include`
  - `include64/lapif.h`: LAPI's 64-bit FORTRAN header file
  - `lib/liblapi_r.a`: LAPI's library file
  - `libtrace/liblapi_r.a`: LAPI's library file with tracing enabled
  - `lib/power6/liblapi_r.a`: LAPI's library file for POWER6 systems
  - `libtrace/power6/liblapi_r.a`: LAPI's library file for POWER6 systems with tracing enabled
  - `lib/lapisub.exp`: Subroutine export file, with a link in `/usr/lib`
  - `lib/lapisub64.exp`: Subroutine export file (64-bit version), with a link in `/usr/lib`
  - `pnsd/lib/libpnsd.a`: PNSD library file, with a link in `/usr/lib`
  - `pnsd/lib/libnrt.a`: NRT library file, with a link in `/usr/lib`
  - `pnsd/include/nrt.h`: NRT header file, with a link in `/usr/include`
  - `dev/include/css_shared.h`: Header file that contains communication subsystem (CSS) information (required by LAPI)

  The following files are installed directly into the `/usr/lib` tree:
drivers/zcmem_ke
Contains the kernel services that LAPI uses for shared memory

methods/cfgzcmem
Loads the shared memory kernel extension (binary file). AIX is automatically updated so that the configuration is done at boot time.

methods/ucfgzcmem
Unloads the shared memory kernel extension (binary file). AIX is automatically updated so that the configuration is done at boot time.

These files provide LAPI's message catalog for AIX 5.3 and 6.1:

- nls/msg/C/liblapi.cat
- nls/msg/ca_ES/liblapi.cat
- nls/msg/cs_CZ/liblapi.cat
- nls/msg/de_DE/liblapi.cat
- nls/msg/en_US/liblapi.cat
- nls/msg/EN_US/liblapi.cat
- nls/msg/es_ES/liblapi.cat
- nls/msg/fr_FR/liblapi.cat
- nls/msg/hu_HU/liblapi.cat
- nls/msg/it_IT/liblapi.cat
- nls/msg/ja_JP/liblapi.cat
- nls/msg/ko_KR/liblapi.cat
- nls/msg/pl_PL/liblapi.cat
- nls/msg/pt_BR/liblapi.cat
- nls/msg/ru_RU/liblapi.cat
- nls/msg/sk_SK/liblapi.cat
- nls/msg/zh_CN/liblapi.cat
- nls/msg/zh_TW/liblapi.cat

rsct.lapi.samp
Contains LAPI's sample files. These files are not required, but are recommended. LAPI has an extensive set of sample files that demonstrate various aspects of the API, including new features. See Chapter 22, "Sample LAPI programs," on page 291 for more information.

### Installation steps

1. If you have a previous version of RSCT LAPI for AIX installed, you can optionally uninstall the rsct.lapi.nam fileset before you install the current version of RSCT LAPI for AIX. Use the AIX `installp` command to uninstall
   \[ `installp -u rsct.lapi.nam` \]

2. Use the AIX `installp` command to install the LAPI filesets. For example, to install all of the LAPI filesets, enter:
   \[ `installp -a -d rsct.lapi all` \]

3. If you plan to use the rsct.lapi.nam fileset for failover and recovery, perform this step:
Reboot the system after the installation is finished.

See the appropriate AIX Commands Reference for more information on installp.

---

**Post-installation steps**

1. If you installed the `rsct.lapi.bsr` fileset, you need to enable the BSR through the Hardware Management Console (HMC). System i and System p: Partitioning for AIX with an HMC describes how you can assign BSR to logical partitions using version 7 (or later) of the HMC.

---

**Uninstallation steps**

1. Perform this step if you have the `rsct.lapi.nam` fileset installed on your system.
   a. If you are running RSCT group services, use the `stopsrc` command to stop `cthagslsm`:
      ```bash
      stopsrc -s cthagslsm
      ```
      See the appropriate AIX Commands Reference for more information on `stopsrc`.
   b. Run the following command to remove the NAM pseudo-device (`nampd0`):
      ```bash
      /usr/bin/rmdev -d -l nampd0
      ```
      For more information about NAM, see [The NAM component of RSCT LAPI for AIX](#) on page 128. For more information about `rmdev`, see the appropriate AIX Commands Reference.

2. Perform this step if you have the `rsct.lapi.bsr` fileset installed on your system.
   a. Run the following command to remove the BSR pseudo-device (`bsrp0d0`):
      ```bash
      /usr/bin/rmdev -d -l bsrp0d0
      ```

3. Use the `installp` command to uninstall the LAPI filesets. For example, to uninstall all of the LAPI filesets, enter:
   ```bash
   installp -u rsct.lapi
   ```
   See the appropriate AIX Commands Reference for more information on `installp`. 
Chapter 5. Installing LAPI for Linux

This chapter includes information about installing version 3.1.3.0 of LAPI for Linux, which is shipped with the Parallel Environment (PE) for Linux licensed program.

For information about installing PE for Linux, see PE: Installation.

For information about installing TWS LoadLeveler for Linux, see TWS LoadLeveler: Installation Guide.

For information about installing RSCT LAPI for AIX, see Chapter 4, “Installing RSCT LAPI for AIX,” on page 29.

Requirements

To make use of all of LAPI’s functions, you must make sure that the hardware and software requirements for your particular Linux environment are satisfied.

Hardware

In order to take advantage of all of the functions of LAPI for Linux, you must be using the following hardware:

• One of the following servers:
  – An IBM BladeCenter (Power Architecture or X-Architecture) server
  – An IBM Power Systems server
  – An IBM System x server

• One of the following adapters, if you are running over IP:
  – 10/100 Ethernet adapter.
  – 1 GB Ethernet adapter.
  – 1 GB Myrinet switch adapter.
  – GX Dual-Port 4x InfiniBand Host Channel Adapter (HCA). AIX 5.3 with the 5300-03 Recommended Maintenance Package (or later), running on an IBM System p5 505 node, is required.
  – Mellanox ConnectX dual port DDR IB 4X HCA PCIe 2.0 x8 adapter.
  – Topspin adapter over InfiniBand. AIX 5.3 with the 5300-03 Recommended Maintenance Package (or later), running on a 64-bit POWER4 (or greater) node, is required.
  – Channel bonding configuration support (an arrangement in which two or more network interfaces on a host computer are combined for redundancy or increased throughput).

Software

To use all of the available features and functions of LAPI for Linux®, you must have the following software installed on your system:

• One of the following Linux distributions, if you are running over IP:
  – Red Hat Enterprise Linux (EL) 5 Update 2 (or later).
  – SUSE LINUX Enterprise Server (SLES) 10 (or later).

• Version 5.1 (or later) of the Parallel Environment for Linux licensed program (product number 5765-PEL), which include the LAPI for Linux RPMs
• Version 3.5 (or later) of the TWS LoadLeveler licensed program (product number 5765-D61).

• One of the following C, C++, or FORTRAN compilers:
  An IBM compiler for Linux, for use on IBM BladeCenter, IBM eServer OpenPower, and IBM System p servers:
  – IBM C compiler, Version 7.0.1-0 (or later) (vac.cmp-7.0.1-0) (on each node)
  – IBM C++ compiler, Version 7.0.1-0 (or later) (vacpp.cmp-7.0.1-0) (on each node)
  – IBM XL FORTRAN compiler, Version 12 (or later) (xlf.cmp-12.0.0-0)
  A GNU compiler for Linux (from the Free Software Foundation), for use on all servers that support PE for Linux:
  – GNU Compiler Collection (GCC) C compiler, Version 3.4.4-2 (or later) (gcc-3.4.4-2) (on each node)
  – GCC C++ compiler, Version 3.4.4-2 (or later) (gcc-c++-3.4.4-2) (on each node)
  – For compiling FORTRAN 77 programs: GCC FORTRAN 77 compiler, Version 3.4.4-2 (gcc-g77-3.4.4-2)

How is LAPI for Linux packaged?

The LAPI RPMs are included on the PE for Linux product CD. See PE: Installation for more information.

Table 14 lists the LAPI RPMs required for installation, based on your platform. In general, install the base LAPI RPM first, then install the related LAPI RPMs.

Table 14. LAPI RPMs required for installation

<table>
<thead>
<tr>
<th>Platforms</th>
<th>RPMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red Hat EL 5 running on:</td>
<td>lapi_ppc_32bit_base_IP_rh500-3.1.3.0-build-level.ppc.rpm</td>
</tr>
<tr>
<td>• IBM BladeCenter servers</td>
<td>lapi_ppc_64bit_IP_rh500-3.1.3.0-build-level.ppc64.rpm</td>
</tr>
<tr>
<td>• IBM eServer OpenPower servers</td>
<td></td>
</tr>
<tr>
<td>• IBM System p servers</td>
<td></td>
</tr>
<tr>
<td>• IBM System p5 servers</td>
<td></td>
</tr>
<tr>
<td>Red Hat EL 5 running on:</td>
<td>lapi_x86_32bit_base_IP_rh500-3.1.3.0-build-level.x86.rpm</td>
</tr>
<tr>
<td>• IBM eServer 325, 326 servers</td>
<td>lapi_x86_64bit_IP_rh500-3.1.3.0-build-level.x86_64.rpm</td>
</tr>
<tr>
<td>• 64-bit IBM System x servers</td>
<td></td>
</tr>
<tr>
<td>Red Hat EL 5 running on:</td>
<td>lapi_x86_32bit_base_IP_rh500-3.1.3.1.3.0-build-level.x86.rpm</td>
</tr>
<tr>
<td>• 32-bit IBM System x servers</td>
<td></td>
</tr>
<tr>
<td>SLES 10 running on:</td>
<td>lapi_ppc_32bit_base_IP_sles1000-3.1.3.0-build-level.ppc.rpm</td>
</tr>
<tr>
<td>• IBM BladeCenter servers</td>
<td>lapi_ppc_64bit_IP_sles1000-3.1.3.0-build-level.ppc64.rpm</td>
</tr>
<tr>
<td>• IBM eServer OpenPower servers</td>
<td></td>
</tr>
<tr>
<td>• IBM System p servers</td>
<td></td>
</tr>
<tr>
<td>• IBM System p5 servers</td>
<td></td>
</tr>
</tbody>
</table>
Table 14. LAPI RPMs required for installation (continued)

<table>
<thead>
<tr>
<th>Platforms</th>
<th>RPMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLES 10 running on:</td>
<td></td>
</tr>
<tr>
<td>• IBM eServer 325, 326 servers</td>
<td>lapi_x86_32bit_base_IP_sles1000-3.1.3.0-build-level.x86.rpm</td>
</tr>
<tr>
<td></td>
<td>lapi_x86_64bit_IP_sles1000-3.1.3.0-build-level.x86_64.rpm</td>
</tr>
<tr>
<td>• 64-bit IBM System x servers</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>SLES 10 running on:</td>
<td></td>
</tr>
<tr>
<td>• 32-bit IBM System x servers</td>
<td>lapi_x86_32bit_base_IP_sles1000-3.1.3.0-build-level.x86.rpm</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>SLES 11 running on:</td>
<td></td>
</tr>
<tr>
<td>• IBM BladeCenter servers</td>
<td>lapi_ppc_32bit_base_IP_sles1100-3.1.3.0-build-level.ppc.rpm</td>
</tr>
<tr>
<td>• IBM eServer OpenPower servers</td>
<td>lapi_ppc_64bit_IP_sles1100-3.1.3.0-build-level.ppc64.rpm</td>
</tr>
<tr>
<td>• IBM System p servers</td>
<td></td>
</tr>
<tr>
<td>• IBM System p5 servers</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>SLES 11 running on:</td>
<td></td>
</tr>
<tr>
<td>• IBM eServer 325, 326 servers</td>
<td>lapi_x86_32bit_base_IP_sles1100-3.1.3.0-build-level.x86.rpm</td>
</tr>
<tr>
<td></td>
<td>lapi_x86_64bit_IP_sles1100-3.1.3.0-build-level.x86_64.rpm</td>
</tr>
<tr>
<td>• 64-bit IBM System x servers</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>SLES 11 running on:</td>
<td></td>
</tr>
<tr>
<td>• 32-bit IBM System x servers</td>
<td>lapi_x86_32bit_base_IP_sles1100-3.1.3.0-build-level.x86.rpm</td>
</tr>
</tbody>
</table>

The LAPI RPMs contain the libraries, scripts, sample programs, messages, and various other files associated with LAPI.

The following is a list of files and directories that are either created or installed as a result of installing the LAPI RPMs:

Table 15. Directories and files associated with the LAPI RPMs

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/opt/ibmhpc/lapi/include</td>
<td>Contains LAPI's 32-bit header files.</td>
</tr>
<tr>
<td>/opt/ibmhpc/lapi/include64</td>
<td>Contains LAPI's 64-bit header files.</td>
</tr>
<tr>
<td>/opt/ibmhpc/lapi/lib</td>
<td>Contains the LAPI libraries.</td>
</tr>
<tr>
<td>/opt/ibmhpc/lapi/msg</td>
<td>Contains the LAPI message catalogs.</td>
</tr>
<tr>
<td>/opt/ibmhpc/lapi/samples</td>
<td>Contains the LAPI sample programs.</td>
</tr>
</tbody>
</table>

These files are linked to /opt/ibmhpc:

- /usr/lib/liblapi.so
- /usr/lib64/liblapi.so
- /usr/lib/liblapiudp.so
- /usr/lib64/liblapiudp.so

For information about:

- PE RPMs, see *PE: Installation*
- TWS LoadLeveler RPMs, see *TWS LoadLeveler: Installation Guide*
Installation steps

If you are using LAPI on a standalone system, follow these installation steps. Otherwise, follow the LAPI installation steps in PE: Installation.

To install the LAPI RPMs:
1. Log in as root.
2. Use the `rpm -i` command to install the required RPMs. For example, enter:

   ```
   rpm -i lapi_x86_32bit_base_IP_rh500-3.1.3.0-build-level.x86.rpm
   rpm -i lapi_x86_64bit_IP_rh500-3.1.3.0-build-level.x86_64.rpm
   rpm -i lapi_x86_32bit_US_rh500-3.1.3.0-build-level.x86.rpm
   rpm -i lapi_x86_64bit_US_rh500-3.1.3.0-build-level.x86_64.rpm
   ```

Uninstallation steps

To uninstall the LAPI RPMs:
1. Login as root.
2. Use the `rpm -e` command to uninstall the RPMs in the opposite order you installed them. For example, enter:

   ```
   rpm -e lapi_x86_64bit_US_rh500-3.1.3.0-build-level.x86_64.rpm
   rpm -e lapi_x86_32bit_US_rh500-3.1.3.0-build-level.x86.rpm
   rpm -e lapi_x86_64bit_IP_rh500-3.1.3.0-build-level.x86_64.rpm
   rpm -e lapi_x86_32bit_base_IP_rh500-3.1.3.0-build-level.x86.rpm
   ```
Chapter 6. Setting up, initializing, and terminating LAPI

This chapter explains how to set up, initialize, and terminate LAPI on systems running IBM's Parallel Environment (PE) licensed program and on standalone systems (those not running PE). For specific information that applies only to the use of LAPI in standalone mode, see Chapter 17, "Using LAPI on a standalone system," on page 141.

Setting and querying the LAPI environment

LAPI allows users to configure the LAPI environment using two different types of variables: environment variables and runtime attributes.

*Environment variables* are set before job initialization. See "Environment variables" on page 317 for more information.

*Runtime attributes* can be retrieved using the `LAPI_Qenv` subroutine and set using the `LAPI_Senv` subroutine at runtime. See "LAPI_Qenv" on page 221, "LAPI_Senv" on page 234, and "Runtime attributes" on page 325 for more information.

Setting environment variables

You need to set one or more environment variables to use LAPI.

**Required environment variables**

You must set the `MP_MSG_API` environment variable before LAPI is initialized. The format of the `MP_MSG_API` specification is:

```
MP_MSG_API="api1[api2]...[(count1)],[api3[api4]...[(count2)]]]...
```

where:

- **api** Specifies a parallel API (or protocol) name: `lapi`, `mpi`, or a name you define that is passed to `LAPI_Init`.
- **count** Specifies the repeat count of the `api[api]...` parameter. If no repeat count is specified, a value of 1 is assumed.

`lapi` and `mpi` are reserved parallel API names. `mpi` has only a single LAPI context. User-defined parallel API names can contain alphanumeric characters and are not case sensitive. You can specify up to 14 parallel API names. One parallel API name can appear only once.

The elements in the specification are separated by commas and each element is mapped to its corresponding LAPI context. In each element, a set of parallel APIs that is separated by an underscore (_) shares the same LAPI context. To repeat a specification, include an integer in parentheses after the parallel API names and enclose the entire specification in double quotation marks. If a repeat count in parentheses follows a parallel API name, the parallel API is repeated that many times. For example:

```
MP_MSG_API="mpi,upc_lapi(8)"
```

indicates that there is one LAPI context for `mpi` and that there are eight LAPI contexts to be shared between `lapi` and `upc`. 

© Copyright IBM Corp. 2003, 2009
The maximum number of LAPI contexts in a task is 128, as up to 128 tasks are supported on a node. When all of the resources on the node are allocated for one task, the task can have up to 128 LAPI contexts.

When two or more parallel APIs share a single context, the second parallel API cannot be initialized after the first parallel API has been terminated.

If more parallel APIs than are actually used by the job are specified, the job can run correctly, but some resources are wasted, that is, held by the job and not used until the job is terminated. If fewer parallel APIs than are actually used by the job are specified, the job will fail with an error during initialization.

For users running POE, the default setting for this environment variable is mpi. This setting applies to MPI communication only; it does not apply to LAPI communication. If you are using LAPI, you cannot use the default setting, as LAPI will not initialize if MP_MSG_API=mpi is set. You must set this environment variable explicitly to one of the settings that allows for LAPI communication. In a LAPI environment, valid settings for MP_MSG_API include the following:

- lapi: Sets up LAPI communication using an exclusive adapter window. Tasks can communicate using only LAPI calls.
- lapi,mpi | mpi,lapi: Sets up LAPI and MPI communication in the same job using separate adapter windows. Both protocols are used, with dedicated resources assigned to each of them. Tasks can communicate using LAPI calls or MPI calls.
- lapi_mpi | mpi_lapi: Sets up LAPI and MPI communication in the same job using a shared adapter window. Both protocols are used, sharing the same set of communication resources (windows, IP addresses) between them. LAPI tasks and MPI tasks communicate using a shared handle.
- api1,api2 | api2,api1: Indicates that calls two parallel APIs are used in the application, with dedicated contexts (resources) assigned to each of them.
- api1,api2 | api2,api1: Indicates that calls to two parallel APIs are used in the application and share a single LAPI context (set of communication resources) between them.

You can combine specifications and add additional parallel API names and repeat counts as needed for your environment.

Optional environment variables

The following environment variables are also commonly used:

- LAPI_USE_SHM=[yes|no|only] (no is the default)
- MP_EUILIB=[ip|us] (ip is the default)
- MP_PROCS=number_of_tasks_in_job

See “Environment variables” on page 317 for more information.
**LAPI communication modes**

LAPI communication takes place in one of the following modes:

- User space (US) over the HPS (for RSCT LAPI users).
- User Datagram Protocol / Internet Protocol (UDP/IP) over the HPS (for RSCT LAPI users) or any other device that supports IP communication (for all LAPI users).
- Shared memory, for tasks that are running on the same node (for all LAPI users).

This document refers to such tasks as *common tasks*.

For a given job, US and UDP/IP communication are mutually exclusive.

It is also possible to combine shared memory mode with one of the other two modes. There are a number of possibilities:

1. If **LAPI_USE_SHM=no** (or is not set, as no is the default) and:
   a. **MP_EUILIB=ip** (or is not set, as ip is the default), LAPI sets up all tasks to communicate using UDP/IP.
   b. **MP_EUILIB=us**, LAPI sets up all tasks to communicate using the user space (US) protocol.
2. If **LAPI_USE_SHM=yes**, an attempt is made to initialize shared memory among all common tasks. In addition, LAPI sets up communication based on the setting of **MP_EUILIB** as described in 1a and 1b above. This is the mode of communication for any tasks:
   - that are not on the same node
   - for which shared memory setup fails
3. If **LAPI_USE_SHM=only** and:
   a. All tasks are common, LAPI attempts to set up shared memory communication among all tasks. If this attempt fails, **LAPI_Init** returns an error.
   b. All tasks are not common (the job is spread across multiple nodes), initialization fails and **LAPI_Init** returns an error.

[Figure 1 on page 42](#) illustrates LAPI's sequence of events for setting up the mode of communication.
Initializing LAPI

To initialize LAPI, follow these steps:

1. Set environment variables (as described in "Setting environment variables" on page 39) before the user application is invoked. The remaining steps are done in the user application.

2. Clear lapi_info_t, then set any fields.
   See "LAPI_Init" on page 199 for the lapi_info_t structure. See "Passing information to LAPI using lapi_info_t" and "Registering an error handler" on page 43 for more information about using this structure.

3. Call LAPI_Init.
   See "LAPI_Init" on page 199 for more information about using LAPI_Init, including various examples of LAPI initialization.

Passing information to LAPI using lapi_info_t
The second argument to LAPI_Init is the address of a lapi_info_t structure. You can use this structure to pass certain information to LAPI. Certain fields in the structure are reserved for future use and should be cleared before calling...
**LAPI_Init.** It is strongly recommended that the entire structure be cleared, and then only desired fields get set. You can clear the memory of a `lapi_info_t` structure using the `bzero` subroutine. For example, suppose you have a structure declared as follows:

```c
lapi_info_t info_struct;
```

You would clear this structure by calling:

```c
bzero(&info_struct,sizeof(lapi_info_t));
```

For more information about `lapi_info_t`, see "LAPI_Init" on page 199. For more information about `bzero`, see *AIX 5L Version 5.3 Technical Reference: Base Operating System and Extensions, Volume 1* or *AIX 5L Version 6.1 Technical Reference: Base Operating System and Extensions, Volume 1*.

**Registering an error handler:** Use the `err_hndlr` field in `lapi_info_t` to register an error handler that you provide. Your error handler is invoked by the occurrence of an asynchronous error. LAPI calls its own internal error handler, which then calls your error handler, if it is registered.

If your error handler is registered, you can control whether execution will continue, based on the type of error that is returned. You can choose to dump information, terminate the job from within its error handler, or both. If the user handler returns, LAPI continues execution. If the error handler you provided is not registered, LAPI’s internal error handler will terminate the job.

Normally, an error handler terminates. Your error handler is intended for the types of actions that are performed by the `atexit` subroutine, so you should do cleanup processing and exit. If you don’t exit, LAPI’s state is undefined. For example, there is no guarantee that outstanding messages will complete.

The format of a user-provided error handler follows:

```c
void my_err_hndlr(lapi_handle_t *hndl, int *error_code, lapi_err_t *err_type,
    int *task_ID, int *src)
{
    char errstring[100]; /* for error code translation */

    /* get the error code string to decipher the error */
    LAPI_Msg_string(*error_code, errstring);
    fprintf(stderr, "%s\n", errstring);

    if ( you want to terminate ) {
        LAPI_Term(*hndl); /* to terminate LAPI */
        exit(some_return_code);
    }

    /* any additional processing */

    return; /* signal to LAPI that the error is not */
    /* fatal and execution should continue */
}
```

```c
/*...

    lapi_handle_t hndl;
    lapi_info_t info;

    bzero(&info, sizeof(lapi_info_t)); /* clear lapi_info */

    /* set the error handler pointer */
    info.err_hndlr = (LAPI_err_hndlr) my_err_hndlr;
```

Chapter 6. Setting up, initializing, and terminating LAPI
LAPI_Init(&hndl, &info);
::
}


Terminating LAPI

Use the LAPI_Term subroutine to terminate a LAPI context that is specified by a particular LAPI handle. For example:
LAPI_Term(hndl);

Any LAPI notification threads that are associated with this context are terminated. If any LAPI calls are made using hndl after LAPI_Term is called, an error occurs. See "LAPI_Term" on page 239 for more information.
Chapter 7. Transferring data

LAPI provides “put”, “get”, and read-modify-write (RMW) functions for data transfer.

LAPI’s “put” and “get” functions are non-blocking calls. A “put” operation copies data from a specified region in the origin address space to the specified region in the target address space. A “get” operation copies data from a specified region in the target address space to a specified region in the origin address space. Completion of the operation is signaled if counters are specified.

For “get” functions, only synchronous operation is possible. For “put” functions, both standard and synchronous operations are supported. A standard “put” operation is provided by incrementing the origin counter (org_cnt) when the origin buffer can be reused. To guarantee the standard behavior of the LAPI “put” operations, you can use the LAPI_Waitcntr subroutine in conjunction with the origin counter (org_cnt). A synchronous “put” operation is provided by incrementing the completion counter (cmpl_cnt) after the data has been written into the target buffer.

To guarantee the synchronous behavior of the LAPI “put” operations, you can use the LAPI_Waitcntr subroutine in conjunction with the completion counter (cmpl_cnt). See “Flow of “put” operations” and “Flow of “get” operations” on page 46 for more information.

The LAPI_Xfer subroutine provides functions that are similar to LAPI_Put and LAPI_Get, with some enhancements. See “LAPI_Xfer” on page 260 for more information.

For RMW information, see “Flow of read-modify-write operations” on page 46.

For systems running over US, RSCT LAPI for AIX supports bulk message transfer using the remote direct memory access (RDMA) protocol. See “Bulk message transfer on AIX” on page 63 for more information.

For HPS systems, RSCT LAPI for AIX supports user-initiated RDMA operations. See “User-initiated RDMA transfer on AIX” on page 65 for more information.

Data transfer operations

Information about the flow of LAPI’s data transfer operations follows.

Flow of “put” operations

Figure 2 on page 46 illustrates the sequence of events for a LAPI_Put, LAPI_Putv, PUT, or PUTV operation. If an origin counter is specified, it is incremented by LAPI on the sending side when the send-side data buffer is available for reuse. If a target counter is specified, it is incremented by LAPI on the target side upon message completion. If a completion counter is specified, LAPI sends an internal message to the original sender. Upon receipt of this message, the completion counter is incremented by LAPI at the origin.
Flow of “get” operations

Figure 3 illustrates the sequence of events for a LAPI_Get, LAPI_Getv, GET, or GETV operation. In this case, the “source” is actually the receiver of the message, and the target can be thought of as the “sender”. Because the direction of the data transfer is from the target to the source, it is helpful to think of the origin and target counters switching roles from the “put” case. If an origin counter is specified, it is incremented on the source task when the message completes. If a target counter is specified, it is incremented on the target side once the target data buffer is available (that is, when its contents can be modified without corrupting the message in transit). Because the origin counter actually signifies message completion, there is no completion counter for “get” operations.

Flow of read-modify-write operations

Figure 4 on page 47 illustrates the sequence of events for a LAPI_Rmw operation or a LAPI_Rmw64 operation. Only the origin counter is supported, and its behavior...
is similar to that of the origin counter in the **LAPI_Get** case.

---

**Non-contiguous data transfer**

LAPI supports two types of non-contiguous data transfer: LAPI vectors and data gather/scatter programs (DGSPs). Use LAPI vectors when the data to be transferred can be described simply, such as through the explicit specification of multiple buffer address/length pairs, or through the repetition of a data template that is described by a block size and stride. For more complex transfers, such as through recursive data descriptions, use one or more DGSPs.

**Using vectors**

LAPI vector transfers involve the specification of an origin vector description and a target vector description, each of type `lapi_vec_t`. The API calls for vector data transfer are: `LAPI_Amsendv`, `LAPI_Getv`, `LAPI_Putv`, `AMV`, `GETV`, and `PUTV`. These APIs are vector-based versions of the basic API calls of the same name: `LAPI_Amsend`, `LAPI_Get`, `LAPI_Put`, `AM`, `GET`, and `PUT`. For instance, `LAPI_Amsendv` can be thought of as "LAPI_Amsend with vectors." This document refers to the three non-vector calls (LAPI_Amsend, LAPI_Get, and LAPI_Put) as **basic calls** and the type of transfer that is associated with these basic calls as **scalar transfer**. The corresponding LAPI vector calls (LAPI_Amsendv, LAPI_Getv, and LAPI_Putv) are referred to as **vector calls** and the type of transfer that is associated with these calls as **vector transfer**.

LAPI vector transfers are set up similarly to scalar transfers. The difference lies in the data specification. For a basic transfer, an origin address, target address, and data length are required. For a vector transfer, two vector descriptions are required — one for the origin and one for the target. Consider the basic **LAPI_Amsend** call. The origin task provides an origin buffer (base address and data length) and the address or index of a header handler on the target task. The target header handler is expected to return a buffer address on the target task, using the origin data length as part of the target buffer definition. A data buffer in LAPI is defined by a base address and a data length. For **LAPI_Amsendv**, the origin task is required to pass in an origin vector description (instead of buffer address and data length) and...
a header handler address or index. Similarly, the target header handler is expected to return the address of a target vector description, rather than a single data buffer.

As another example, when using `LAPI_Get`, you need to specify an origin address, a target address, and a data length. When using `LAPI_Getv`, you need to specify an origin vector description and a target vector description. Using vector descriptions gives you additional flexibility in how the data is collected on the origin side and distributed on the target side. In all vector transfers, the type of vector to be transferred determines the requirements on the origin and target vectors.

LAPI vectors are structures of type `lapi_vec_t`, defined as follows:

```c
typedef struct {
    lapi_vectype_t vec_type;
    uint num_vecs;
    void **info;
    ulong *len;
} lapi_vec_t;
```

`vec_type` is an enumeration that describes the type of vector transfer, which can be one of the following:

- **LAPI_GEN_GENERIC**
  - Supports explicit specification of the data buffers. The number of buffers and the buffer lengths between the origin side and the target side do not need to match; however, the size of the target side must be greater than or equal to the size of the origin side. You can only use this datatype with `LAPI_Amsendv` and `AMV`. `LAPI_GEN_GENERIC` cannot be used with `LAPI_Getv`, `LAPI_Putv`, `GETV`, or `PUTV`.

- **LAPI_GEN_IOVECTOR**
  - Supports explicit specification of the data buffers on the origin side and the target side. The number of buffers in the origin vector description and in the target vector description must be the same. The lengths of the origin vector's data buffers must equal the lengths of the target vector's data buffers.

- **LAPI_GEN_STRIDED_XFER**
  - Supports the transfer of data through one or more iterations of sending a block and offsetting by its stride. The data is described by a block size, stride, and number of invocations. The origin vector description and the target vector description must be the same, though the strides can differ.

If `vec_type` is `LAPI_GEN_GENERIC` or `LAPI_GEN_IOVECTOR`, the fields are used as follows:

- `num_vecs` indicates the number of data vectors to transfer. Each data vector is defined by a base address and data length.
- `info` is the array of addresses.
- `len` is the array of data lengths.

For example, consider the following vector description:

```c
vec_type = LAPI_GEN_IOVECTOR
num_vecs = 3
info = {addr_0, addr_1, addr_2}
len = {len_0, len_1, len_2}
```
For an origin vector, LAPI would read \textit{len\_0} bytes from \textit{addr\_0}, \textit{len\_1} bytes from \textit{addr\_1}, and \textit{len\_2} bytes from \textit{addr\_2}. For a target vector, LAPI would write \textit{len\_0} bytes to \textit{addr\_0}, \textit{len\_1} bytes to \textit{addr\_1}, and \textit{len\_2} bytes to \textit{addr\_2}.

Recall that vector transfers require an origin vector and a target vector. For \texttt{LAPI\_Amsendv} calls, the origin vector is passed to the API call on the origin task. The address of the target vector is returned by the target header handler.

\textbf{LAPI\_GEN\_GENERIC}

For transfers of type \texttt{LAPI\_GEN\_GENERIC}, the target vector description must also have type \texttt{LAPI\_GEN\_GENERIC}. Use this datatype only with \texttt{LAPI\_Amsendv} and \texttt{AMV}. The contents of the \textit{info} and \textit{len} arrays are unrestricted in the generic case. The number of vectors and the length of vectors on the origin and target do not need to match; however, the size of the target side must be greater than or equal to the size of the origin side. In this case, LAPI transfers a given number of bytes in noncontiguous buffers specified by the origin vector to a set of noncontiguous buffers specified by the target vector.

If \texttt{TGT\_LEN} is greater than \texttt{ORG\_LEN}, all \texttt{ORG\_LEN} bytes are transferred. The rest of the target area is left alone. Consider the following example:

\begin{verbatim}
Origin_vector: {
    num_vecs = 3;
    info = {orgaddr\_0, orgaddr\_1, orgaddr\_2};
    len = {5, 10, 5}
}

Target_vector: {
    num_vecs = 4;
    info = {tgtaddr\_0, tgtaddr\_1, tgtaddr\_2, tgtaddr\_3};
    len = {12, 2, 4, 2}
}
\end{verbatim}

LAPI copies data as follows:
1. 5 bytes from \textit{orgaddr\_0} to \textit{tgtaddr\_0} (leaves 7 bytes of space at a 5-byte offset from \textit{tgtaddr\_0})
2. 7 bytes from \textit{orgaddr\_1} to remaining space in \textit{tgtaddr\_0}, which leaves 3 bytes of data to transfer from \textit{orgaddr\_1}
3. 2 bytes from \textit{orgaddr\_1} to \textit{tgtaddr\_1}, which leaves 1 byte to transfer from \textit{orgaddr\_1}
4. 1 byte from \textit{orgaddr\_1} followed by 3 bytes from \textit{orgaddr\_2} to \textit{tgtaddr\_2}, which leaves 2 bytes to transfer from \textit{orgaddr\_2}
5. 2 bytes from \textit{orgaddr\_2} to \textit{tgtaddr\_3}

LAPI copies data from the origin until the space described by the target is filled. For example:

\begin{verbatim}
Origin_vector: {
    num_vecs = 1;
    info = {orgaddr\_0};
    len = {20}
}

Target_vector: {
    num_vecs = 2;
    info = {tgtaddr\_0, tgtaddr\_1};
    len = {5, 10}
}
\end{verbatim}
LAPI copies 5 bytes from orgaddr_0 to tgtaddr_0 and the next 10 bytes from orgaddr_0 to tgtaddr_1. The remaining 5 bytes from orgaddr_0 are not copied.

**LAPI\_GEN\_IOVECTOR**

For transfers of type **LAPI\_GEN\_IOVECTOR**, the lengths of the vectors must match and the target vector description must match the origin vector description. More specifically, the target vector description must:

- Also have type **LAPI\_GEN\_IOVECTOR**.
- Have the same *num_vecs* as the origin vector.
- Initialize the *info* array with *num_vecs* addresses in the target address space.
- Initialize the *len* array with *num_vecs* lengths. The values of those lengths must be the same as the values in the *len* array of the origin vector. In other words, for origin vector *o_vec* and target vector *t_vec*, *o_vec.len[i]* must equal *t_vec.len[i]* for all *i* (0 <= i < *num_vecs*).

For LAPI vectors *origin_vec* and *target_vec* described similarly to the example above, data is copied as follows: for all *i*, *origin_vec.len[i]* bytes are transferred from the address at *origin_vec.info[i]* to the address at *target_vec.info[i]*.

**Figure 5** shows how LAPI transfers data using type **LAPI\_GEN\_IOVECTOR**:

In this case, four transfers take place:

1. *d0* bytes are transferred from *a0* on the origin task to *b0* on the target task
2. *d1* bytes are transferred from *a1* on the origin task to *b1* on the target task
3. *d2* bytes are transferred from *a2* on the origin task to *b2* on the target task
4. *d3* bytes are transferred from *a3* on the origin task to *b3* on the target task
**LAPI_GEN_STRIDED_XFER**

For transfers of type **LAPI_GEN_STRIDED_XFER**:

- the target vector description must match the origin vector description (though the strides can differ)
- the *info* array of the origin and target vectors is used differently than it is for non-strided vector data transfer:

Rather than specifying the set of addresses, the *info* array of the origin and target vectors is used to specify a data block "template", consisting of a base address, block size, and stride. LAPI thus expects the *info* array to contain three integers. The first integer contains the base address, the second integer contains the block size to copy, and the third integer contains the byte stride. In this case, *num_vecs* indicates the number of blocks of data that LAPI should copy, where the first block begins at the base address. The number of bytes to copy in each block is given by the block size and the starting address for all but the first block is given by previous address + stride. The total amount of data to be copied will be *num_vecs* * block size. The *len* field of the vector description structure is not used for **LAPI_GEN_STRIDED_XFER**, so any values it contains are ignored. Consider the following example:

```
Origin_vector: {
    num_vecs = 3;
    info  = [orgaddr, 5, 8]
}
```

Based on this description, LAPI will transfer 5 bytes from orgaddr, 5 bytes from orgaddr+8 and 5 bytes from orgaddr+16.

**Figure 6** shows how LAPI transfers data using type **LAPI_GEN_STRIDED_XFER**:

From the vector descriptions represented in **Figure 6**

![Figure 6: Transferring data with type LAPI_GEN_STRIDED_XFER](image)

1. *o_blk_sz* bytes are transferred from A on the origin task to B on the origin task
2. *o_blk_sz* bytes are transferred from A+*o_stride* on the origin task to B+*t_stride* on the target task
3. *o_blk_sz* bytes are transferred from A+(2*o_stride) on the origin task to B+(2*t_stride) on the target task
Vector data transfer summary

Table 16 summarizes the rules for transferring data using vectors.

<table>
<thead>
<tr>
<th>Type of transfer</th>
<th>Data is specified...</th>
<th>Target vector rules</th>
<th>The origin vector resides:</th>
<th>The target vector resides:</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPI_GEN_GENERIC</td>
<td>By explicit lists of origin and target buffers.</td>
<td>The <em>vec_type</em> field must match the origin vector type.</td>
<td>On the origin for LAPI_Amsendv only.</td>
<td>On the target for LAPI_Amsendv only. The addresses in the <em>info</em> array should be in the target address space.</td>
</tr>
<tr>
<td>LAPI_GEN_IOVECTOR</td>
<td>By explicit lists of origin and target buffers.</td>
<td>The <em>vec_type</em> and <em>num_vecs</em> fields and values in the <em>len</em> array must match the origin vector type.</td>
<td>On the origin for all vector API calls (LAPI_Amsendv, LAPI_Getv, and LAPI_Putv).</td>
<td>On the origin for LAPI_Getv and LAPI_Putv. On the target for LAPI_Amsendv. The addresses in the <em>info</em> array should be in the target address space.</td>
</tr>
<tr>
<td>LAPI_GEN_STRIDED_XFER</td>
<td>With a block size, stride size, and the number of blocks to transfer.</td>
<td>The <em>vec_type</em> field and the contents of the <em>info</em> array must match the origin vector type (though the strides can differ).</td>
<td>On the origin for all vector API calls (LAPI_Amsendv, LAPI_Getv, and LAPI_Putv).</td>
<td>On the origin for LAPI_Getv and LAPI_Putv. On the target for LAPI_Amsendv.</td>
</tr>
</tbody>
</table>

Using data gather/scatter programs (DGSPs)

LAPI supports the data gather/scatter program (DGSP) as a mechanism for describing data layouts in memory so certain LAPI functions can operate directly on non-contiguous user buffers. This support is motivated by the need to support MPI datatype constructors. The MPI standard defines an API for specifying any possible data layout with a user-defined datatype. **MPI_Send** gathers data according to the datatype definition and transmits this data over the network. Likewise, an **MPI_Recv** can accept data from the network and scatter it to any layout in memory under control of an MPI datatype. The DGSP provides a compact and complete method of implementing any data layout an MPI user can specify using the MPI datatype constructors. The MPI library incorporates a “compiler” that is run at an **MPI_Type_commit** call to construct the DGSP that corresponds to the user-defined datatype. Like computer object code, the DGSP is not intended as a human-readable specification.

It is possible for LAPI users to construct a DGSP by hand, just as it is possible for someone to write a computer program directly in object code. For anything with significant complexity, it is expected that the LAPI application that uses DGSP will incorporate some form of “code generator” that can generate DGSP code from some layout specification appropriate to the application data structures. This section documents the data gather/scatter instruction set with which LAPI applications can create DGSP codes. A DGSP consists of a descriptor and a “code” array of integers. The instruction counter (IC) of the DGSM, which is the DGSP interpreter.
within LAPI, is an index into this array of integers. Therefore, the addressing unit of the DGSM is one integer, just as the addressing unit of most computer hardware is one byte.

LAPI provides the **LAPI_VERIFY_DGSP** environment variable as a validation option to help LAPI users who create their own DGSPs and want to validate them as they are registered. Validating these DGSPs will catch many, but not all, possible errors. Validation may especially be needed because multiple DGSP transfers could be in progress concurrently and the DGSM runs to handle these data transfers need to report their failures asynchronously, which could make it difficult for a LAPI user who creates several DGSPs to know which DGSP was running at the failure point. The most common result of an uncatchable error in a user-created DGSP is a segmentation fault, which occurs asynchronously. Debugging a faulty DGSP can be challenging.

Each instruction for the DGSM is encoded as a structure that can be mapped into an integer “code” array. LAPI's header files provide a macro for each instruction that can be used to update the IC as instructions are added to a DGSP. In the example below, a COPY instruction is built and macro **LAPI_DGSM_COPY_SIZE** is used to advance the IC to the proper location for building the next instruction, which will be an ITERATE. You must include an ITERATE instruction in every LAPI DGSP, even if you don't think you'll run the program more than once. The suggested technique for building instructions is to declare a set of pointers, one per instruction type and map the instruction structure into the code array as follows:

```c
lapi_dgsm_copy_t pCopy;
lapi_dgsm_iterate_t pIterate;

ic = 0;
pCopy = (lapi_dgsm_copy_t*) &code[ic];
pCopy->opcode = LAPI_DGSM_COPY;
pCopy->bytes = 5;
pCopy->offset = 0;
ic += LAPI_DGSM_COPY_SIZE;
pIterate = (lapi_dgsm_iterate_t*) &code[ic];
pIterate->opcode = LAPI_DGSM_ITERATE;
```

Each complete DGSP represents the gather or scatter of a specific number of bytes of data. It can be helpful to visualize a DGSP as a sieve or template that can be positioned on a memory buffer to let some byte ranges show through while masking others. A LAPI operation that uses a DGSP also specifies the address of a user buffer and a number of bytes to be processed. If the number of bytes to be processed is greater than the DGSP represents, the DGSP is reinterpreted as many times as needed. The DGSP template is first positioned at offset 0 in the user buffer and is advanced by some stride each additional time it is interpreted. The DGSP descriptor guides this process, so it is essential that the content of the descriptor be understood. This descriptor is represented by the following structure:

```c
typedef enum {LAPI_DGSM_SPARSE=0, LAPI_DGSM_CONTIG,
LAPI_DGSM_UNIT} lapi_dgsp_density_t;

typedef struct {
int code;       /* array containing the DGSP code */
int code_size;  /* size of the DGSP code array */
int depth;      /* required DGSM stack depth */
lapi_dgsp_density_t density;  /* lapi_dgsp_density_t datatype */
long size;      /* datatype packed size */
long extent;    /* datatype extent (stride between reps) */
} lapi_dgsp_descr_t;
```
where:

code

is a pointer to the integer array in which the DGSP has been built.

code_size

is the number of integers in the code array.

depth

specifies the maximum subroutine activation depth in the DGSP. At a minimum, a DGSP has a “main” routine. It can also have "subroutines". The “machine” that interprets the DGSP uses a stack. For a DGSP that has only a “main” routine (because it does not use a GOSUB instruction), depth is equal to 1.

density

is one of three enumerated values:

LAPI_DGSM_SPARSE

indicates that the DGSP data layout has internal gaps.

LAPI_DGSM_CONTIG

indicates that the DGSP data layout is contiguous, but there is a gap either before or after the contiguous section. That is, if the DGSP must be interpreted more than once, the DGSM must deal with a gap.

LAPI_DGSM_UNIT

indicates that the DGSP data layout is contiguous and interpreting the DGSP more than once still represents a contiguous layout.

size

is the number of bytes represented by a single application of the DGSP.

extent

is the stride for repeated interpretation of the DGSP, that is, the number of bytes to advance the template within the user buffer for the next application.

lext

is the offset of the leftmost byte (toward the low address) represented by the DGSP.

rext

is the the offset just past the rightmost byte (toward the high address) represented by the DGSP.

atom_size

indicates that LAPI can choose the number of bytes per packet (if atom_size is set to 0, which is the norm). If atom_size is set to a non-zero value, every packet will contain an integral number of “atoms”.

Consider a C structure that contains one integer followed by one character. The C compiler will put the integer at offset 0 in the structure and the character at offset 4. If an array of such structures is declared, the compiler word-aligns each array element so that there is a 3-byte gap. An efficient DGSP for this structure has two instructions: a COPY and an ITERATE. The code will be discussed in more detail later.

A DGSP descriptor for this structure would be constructed as follows:
int code[LAPI_DGSM_COPY_SIZE+LAPI_DGSM_ITERATE_SIZE];
lapi_dgsp_descr_t dgsp_d;
dgsp_d.code = &code[0];
dgsp_d.code_size = LAPI_DGSM_COPY_SIZE+LAPI_DGSM_ITERATE_SIZE;
dgsp_d.depth = 1;
dgsp_d.density = LAPI_DGSM_CONTIG;
dgsp_d.size = 5;
dgsp_d.extent = 8;
dgsp_d.ext = 0;
dgsp_d.rext = 5;
dgsp_d.atom_size = 0;

A LAPI_Xfer call that used this DGSP to send the first three elements from an
array of such structures would specify array[0] as the buffer address and 15 as the
number of bytes. The DGSP would be interpreted 3 times, first at byte offset 0, then
at byte offset 8, and finally at byte offset 16. The instruction set of the DGSM
contains five instructions, each of which is discussed in some detail below. The first
field in any instruction is the opcode. The other fields are different for each
instruction and are discussed below.

**LAPI_DGSM_COPY**

is the first of two instructions to move data. A COPY instruction represents a
single contiguous block of bytes to be transferred. The instruction fields are
bytes and offset.

**LAPI_DGSM_MCOPY**

is the second of two instructions to move data. An MCOPY instruction is a
variable-length instruction that defines one or more contiguous blocks to
copy. The MCOPY count field specifies how many block descriptions it
contains. Each block description is a block displacement (block_disp, block
length (block_len) tuple. If pMcopy is a pointer to an MCOPY instruction,
block description fields can be identified as:
pMcopy->block[i].block_len and pMcopy->block[i].block_disp.

Because the first block is included in LAPI_DGSM_MCOPY_SIZE, the size
of the instruction is:
LAPI_DGSM_MCOPY_SIZE + (count-1) * LAPI_DGSM_MCOPY_BLOCK_SIZE

**LAPI_DGSM_GOSUB**

calls a DGSP subroutine. The subroutine runs with its own stack frame and
does not change the state of the calling stack frame. The subroutine can be
visualized as a subordinate template to be applied one or more times within
a single application of the containing template. The reps field specifies how
many times, offset specifies the position relative to the current template
base in bytes, and stride specifies how far to advance the subordinate
template after each repetition. The to_loc and ret_loc are both IC-relative
jumps that are based on the instruction counter at the GOSUB instruction.
For a normal return to the instruction that follows the GOSUB instruction,
the ret_loc field is set to LAPI_DGSM_GOSUB_SIZE. The to_loc field is
always set to the distance from the GOSUB opcode to the opcode of the
first instruction of the subroutine.

**LAPI_DGSM_ITERATE**

must terminate each DGSP "main" program as well as any subroutine
within a DGSP. A DGSP with a depth of 1 has a "main" routine, but no
subroutines. ITERATE will decrement the repetitions counter in the current
stack frame. If the counter is not yet zero, ITERATE will branch to its
iter_loc target to interpret the DGSP or subroutine again at its new position
within the buffer. If the counter is 0, the ITERATE pops the stack and
branches to the subroutine return point. The \textit{iter\_loc} value is the distance from the ITERATE opcode to the first opcode of the DGSP or subroutine. The values are normally negative.

**LAPI\_DGSM\_CONTROL**

is only used in a scatter-side DGSP and in conjunction with a data distribution manager (DDM) function. Most DGSPs will have no CONTROL instruction and the DGSM mode is non-CONTROL by default. A CONTROL puts the scatter DGSM in a mode where every copy operation is done by a call to the registered DDM function. The operation and operand field values are saved by the DGSM when a CONTROL instruction is encountered and are passed as parameters to every DDM function call. An example of using a DDM function is to construct an accumulate reduction operation for arriving data. A DDM (reduction) function could be written that does more than one kind of arithmetic operation and can handle more than one kind of operand. These parameters allow the CONTROL instruction to tell the reduction function what kind of operand and operation it is working on. A DDM function which does not require this versatility may ignore these parameters. Any non-negative value for the operation field puts the DGSM into CONTROL mode. An operation field with value \texttt{LAPI\_DGSM\_NO\_CTL} takes the machine out of CONTROL mode. Because a reduction function must operate on full operands, the sender side DGSP must specify an atom size that matches the operand size. If the DDM expects to operate on 16-byte REALs, \texttt{atom\_size} must be 16 so that all transmitted packets contain a multiple of 16 bytes. Also, because LAPI may bypass the scatter DGSM when the data is contiguous, any DGSP that contains a CONTROL instruction must be labeled \texttt{LAPI\_DGSM\_SPARSE} to force the data through the DGSM, which will honor the CONTROL instruction.

See \texttt{[LAPI\_Util] on page 241} for a sample program that creates and registers a DGSP.

**Detecting completion**

LAPI gives users two main approaches for notification of event occurrence during data transmission: handlers and counters. \textit{Handlers} are registered with LAPI as callback pointers to be invoked at certain well-defined points during message transfer. \textit{Counters} are implemented as abstract datatypes and managed through a set of LAPI API calls.

**LAPI handlers**

LAPI uses an active messaging infrastructure. Message retrieval involves the invocation of an asynchronous, active-message handler to process the incoming message. LAPI provides a way for users to register their own receive-side handlers through the \texttt{LAPI\_Amsend} and \texttt{LAPI\_Amsendv} subroutines, as well as through \texttt{LAPI\_Xfer} with transfer types \texttt{LAPI\_AM\_XFER (AM)}, \texttt{LAPI\_AM\_LW\_XFER (AM\_LW)}, \texttt{LAPI\_AMV\_XFER (AMV)}, \texttt{LAPI\_AMX\_XFER (AMX)}, \texttt{LAPI\_AM\_LW\_XFER (AM\_LW)}, and \texttt{LAPI\_DGSP\_XFER (DGSP)}. For certain transfer types, \texttt{LAPI\_Xfer} also provides a send-side handler to run once the send-side data buffer is available for reuse.

LAPI provides three types of handlers that you can register to run at certain points during data transmission:

\textit{Header handler}

Runs on the receiving side upon the arrival of the first message packet. You
need to provide a callback pointer to a header handler function when using the API calls that support it, namely LAPI_Amsend, LAPI_Amsendv, AM_AMLW, AMV, AMX, and DGSP. See [The header handler] for more information.

In addition to the callback pointer, you can optionally provide an index into the header handler function table. See [LAPI_Addr_set] on page 160 for more information.

**Completion handler (or receive completion handler)**
Runs at the target task upon message completion. Its use is optional, and is supported for the same set of API calls as the header handler. LAPI provides a completion handler parameter pointer so you can pass data from the header to the completion handler. The receive completion handler can also be specified in the call to LAPI_Xfer for the LAPI_GET_XFER and LAPI_GETV_XFER transfer types. Recall that in these “get” calls, the receiver is same as the original task that makes the “get” LAPI_Xfer call. See [The completion handler] on page 58 for more information.

**Send completion handler**
Is available only when you use LAPI_Xfer, for all transfer types except AM_AMLW, GET, and GETV. Because LAPI only supports synchronous behavior with its “get” calls, send completion notification is not supported. This handler is invoked on the sending side when the sending data buffer is available for modification. As with the receive completion handler, LAPI provides a parameter pointer with which you can pass information in to the send completion handler, and its use is optional. See [The send completion handler] on page 60 for more information.

**The header handler**
LAPI header handlers are defined to be of the following type:

```c
typedef void * (hdr_hndlr_t)(lapi_handle_t *hndl, void *uhdr, uint *uhdr_len,
                            ulong *msg_len, compl_hndlr_t **comp_h,
                            void **uinfo);
```

The header handler runs on the target task upon the arrival of the first message packet. As shown above, the header handler must return an address value to LAPI. This value is interpreted by LAPI as the base address of the data buffer where it must write the message on the target. Because of this, a header handler must be provided by the user for API calls that support its use. For the LAPI_Amsendv call, the header handler returns a pointer to a lapi_vec_t structure.

Additional information related to the arriving message at the target can be passed back to LAPI through reference parameters of the header handler. For example, if you want to use a completion handler for post-processing of the message after it has completely arrived at the target, a pointer to the completion handler must be returned to LAPI through one of the header handler parameters (denoted by comp_h in the header handler type definition above). Similarly, a parameter to the completion handler is also set and returned by way of another of the header handler parameters (denoted by uinfo in the header handler type definition). Note that the use of a completion handler is optional, so if either of the pointers is not going to be used, you need to set these header handler parameters to NULL within the header handler. Otherwise, LAPI will interpret the value as a valid function callback address and attempt to invoke it on message completion, which will almost certainly lead to a memory fault. You can use a completion handler and still set the parameter pointer to NULL (LAPI will simply invoke the completion handler without any parameters). If you set the completion handler as NULL and assign some non-zero value for the parameter pointer, the parameter pointer will be ignored.
To define a header handler:

1. Without using a completion handler:

```c
void *my_hdr_hndlr(lapi_handle_t *hndl, void *uhdr, uint *uhdr_len,
                    ulong *msg_len, compl_hndlr_t **comp_h, void **uinfo) {
    /* some user code */
    *comp_h = NULL;
    *uinfo = NULL;
    return data_buffer;
}
```

2. Using a completion handler without a parameter:

```c
void *my_hdr_hndlr(lapi_handle_t *hndl, void *uhdr, uint *uhdr_len,
                    ulong *msg_len, compl_hndlr_t **comp_h, void **uinfo) {
    /* pass the completion handler back by reference */
    *comp_h = compl_hndlr;
    *uinfo = NULL;
    return data_buffer;
}
```

For an example that illustrates the use of a completion handler and a completion handler parameter, see "The completion handler." For advanced programming information, see "The enhanced header handler interface" on page 95.

The completion handler

LAPI calls the completion handler after the final packet of the message has been completely received into the target buffers. Unlike header handlers, completion handlers are optional. LAPI completion handlers are defined to be of the following type:

```c
typedef void (compl_hndlr_t)(lapi_handle_t *hndl, void *completion_param);
```

LAPI provides a facility with which you can pass information from the header handler to the completion handler. You set both the completion handler pointer and completion handler parameter pointer in the header handler (returned to LAPI by reference parameters). An example of using a completion handler and completion handler parameter follows. Note that this example uses a user-defined type for the completion parameter. It is possible through C type-casting to send in a single `long` or `int` value through this parameter instead of sending in a pointer to a complex datatype.

To use a completion handler with a completion handler parameter:

```c
/* user-defined structure for data to pass */
typedef struct {
    int a;
    char b;
} user_compl_t;

/* user's completion handler */
void compl_hndlr(lapi_handle_t *hndl, void *completion_param) {
    /*
    ** LAPI passes the uinfo parameter that was returned through the
    ** header handler in to the completion_param argument. We recast
    ** the parameter to be our defined type.
    */
```
user_compl_t *compl_t = (user_compl_t *) completion_param;

/* use data from the parameter, execute other statements, */
/* then free the parameter */
}

/* user's header handler */
void *my_hdr_hndlr(lapi_handle_t *hndl, void *uhdr, uint *uhdr_len,
                   ulong *msg_len, compl_hndlr_t **comp_h,
                   void **uinfo) {
    user_compl_t *compl_t;
    /* malloc and store values in compl_t here; completion handler should free */
    /* this is how reference parameters are used to pass the pointers to LAPI */
    *comp_h = compl_hndlr;
    *uinfo = (void*) compl_t;
    return data_buffer;
}

A common use of the completion handler is to set a user-managed (as opposed to LAPI-managed) counter on which another thread is waiting. For example:

volatile int user_cntr;
:
:

/* user's completion handler */
void compl_hndlr(lapi_handle_t *hndl, void * completion_param)
{
    user_cntr++;
}

/* some other routine */
{
    /* some LAPI calls */
    :
    while ( user_cntr < some_threshold ) {
        /* do other work */
    }
}

Completion handler execution: To ensure progress on other messages during completion handler execution, LAPI maintains a queue of completion handlers, which is serviced by a dedicated completion handler thread. Under normal execution, completion handler pointers are enqueued by the dispatcher at message completion and then executed within this separate thread. See Figure 7 on page 60.
The use of a separate thread allows the dispatcher to make progress on other messages while the completion handler executes. If you set a completion handler using the header handler reference parameter, the LAPI dispatcher will enqueue the completion handler to the completion handler queue upon message completion.

If a number of handlers pile up, there may be some lag in completion handler execution. LAPI provides users with a mechanism for requesting that this process be short-circuited on a per-message basis and that completion handlers be run inline if possible. As discussed in "The enhanced header handler interface" on page 95, this can be done by setting the `ret_flags` field in the `lapi_return_info_t` structure to `LAPI_LOCAL_STATE` or `LAPI_SEND_REPLY`. For `AM_LW`, the completion handler is always executed inline. See "Inline handlers" on page 97 for more information.

The send completion handler
LAPI provides users with the ability to provide a callback for send completion. This handler runs on the sending side. As with the (receive) completion handler, the use of a send completion handler is optional. If a send completion handler is specified by setting the corresponding field in the structure that is passed in to `LAPI_Xfer` to a non-NULL value, LAPI calls this handler when the send-side data buffer is available for reuse. Note that the send completion handler is only available through the `LAPI_Xfer` interface to the API calls. `LAPI_Xfer` also provides a parameter pointer for passing information in to the send completion handler. The handler's signature is as follows:

```c
void send_compl_hdlr(lapi_handle_t *hdl, void *completion_param, lapi_sh_info_t *info);
```

To use the send completion handler in C, set the `shdlr` pointer (and optionally, the `sinfo` pointer) of the `lapi_xfer_t` union of `LAPI_Xfer` to a non-NULL value. For example:

```c
lapi_xfer_t xfer_struct;
some_user_type *info_ptr;
 :
 xfer_struct.Am.shdlr = (scompl_hdlr_t *) send_compl_hdlr;
 xfer_struct.Am.sinfo = (void *) info_ptr;
 :
 LAPI_Xfer(hndl, &xfer_struct);
```


**LAPI handler summary**

Table 17 provides summary information about the various handlers.

<table>
<thead>
<tr>
<th>Type of handler</th>
<th>Resides on:</th>
<th>Which subroutines use it?</th>
<th>How do I specify it?</th>
<th>How is it used?</th>
<th>When does it run?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>The target task</td>
<td>Required for <code>LAPI_Amsend</code>, <code>LAPI_Amsendv</code>, <code>AM</code>, <code>AM_LW</code>, <code>AMV</code>, <code>AMX</code>, and <code>DGSP</code></td>
<td>Pass it as a callback pointer to the API call or (in C) pass it as a member of a <code>lapi_xfer_t</code> union for <code>LAPI_Xfer</code>. In addition to the callback pointer, optionally provide an index in to the header handler function table.</td>
<td>Must return the target-side data buffer address to LAPI. Optionally, sets completion handler and completion handler parameter pointers through reference parameters.</td>
<td>Upon arrival of the first data packet at the target task</td>
</tr>
<tr>
<td>Completion</td>
<td>The origin task</td>
<td>Optional for <code>GET</code> and <code>GETV</code></td>
<td>Callback pointer set as a member of a <code>lapi_xfer_t</code> union for <code>LAPI_Xfer</code> (in C)</td>
<td>Data can be passed through a parameter pointer set as a member of a <code>lapi_xfer_t</code> union for <code>LAPI_Xfer</code> (in C)</td>
<td>Upon arrival of the last packet of the <code>GET</code> or <code>GETV</code> transfer</td>
</tr>
<tr>
<td>Completion</td>
<td>The target task</td>
<td>Optional for <code>LAPI_Amsend</code>, <code>LAPI_Amsendv</code>, <code>AM</code>, <code>AM_LW</code>, <code>AMV</code>, <code>AMX</code>, and <code>DGSP</code></td>
<td>Callback pointer set in the header handler (user’s option)</td>
<td>Data can be passed through a parameter pointer set in the header handler</td>
<td>Upon arrival of final data packet at the target task (message completion)</td>
</tr>
<tr>
<td>Send completion</td>
<td>The origin task</td>
<td>Optional for <code>AM</code>, <code>AMV</code>, <code>AMX</code>, <code>DGSP</code>, <code>PUT</code>, <code>PUTV</code>, <code>RDMA</code>, and <code>RMW</code></td>
<td>Callback pointer set as a member of a <code>lapi_xfer_t</code> union for <code>LAPI_Xfer</code> (in C). User’s option.</td>
<td>Data can be passed through a parameter pointer set as a member of a <code>lapi_xfer_t</code> union for <code>LAPI_Xfer</code> (in C)</td>
<td>Upon availability of the send-side data buffer</td>
</tr>
</tbody>
</table>

**LAPI counters**

LAPI also provides a number of internally-managed counters, with a set of API calls for setting, getting and waiting on counter values. Three types of counters are provided:

**Origin counter**

Resides in the address space of the origin task. LAPI increments this counter when the send-side data buffer is available for reuse.

**Target counter**

Resides in the address space of the target task. LAPI increments this counter upon message completion at the target.

**Completion counter**

Resides in the address space of the sending side. LAPI increments this counter upon message completion at the target. LAPI sends an internal message back to the sender to indicate message completion.

LAPI provides three subroutines for counter manipulation:
LAPI_Getcntr(lapi_handle_t hndl, lapi_cntr_t *cntr, int *val)
Sets val to the value currently stored by cntr.

LAPI_Setcntr(lapi_handle_t hndl, lapi_cntr_t *cntr, int val)
Sets the value of the counter (cntr) to val.

LAPI_Waitcntr(lapi_handle_t hndl, lapi_cntr_t *cntr, int val, int *cur_cntr_val)
Blocks until the value of the counter (cntr) reaches val. LAPI_Waitcntr stores the current value of the counter in cur_cntr_val and decrements the counter's value by val before returning. The final value of the counter after this call does not need to be 0. (Consider, for example, a call to this subroutine after the value of the counter has already reached a value greater than val.)

Table 18 provides summary information about the various counters.

<table>
<thead>
<tr>
<th>Type of counter</th>
<th>Resides on: The origin task</th>
<th>Which subroutines use it?</th>
<th>How do I specify it?</th>
<th>When is the counter incremented?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Origin</td>
<td>The origin task</td>
<td>LAPI_Amsend,</td>
<td>Pass it as an argument to an API call</td>
<td>Upon availability of the origin data buffer.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LAPI_Amsendv,</td>
<td></td>
<td>For LAPI_Get, LAPI_Getv, GET, and GETV: upon completion of message delivery.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LAPI_Get, LAPI_Getv,</td>
<td></td>
<td>For LAPI_Rmw, LAPI_Rmw64, and RMW: upon completion of return message delivery.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LAPI_Put, LAPI_Putv, AM,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AM_LW, AMV, AMX, GET,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GETV, PUT, and PUTV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Target</td>
<td>The target task</td>
<td>LAPI_Amsend,</td>
<td>Pass it as an argument to an API call</td>
<td>Upon completion of message delivery.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LAPI_Amsendv,</td>
<td></td>
<td>For LAPI_Get, LAPI_Getv, GET, or GETV: upon availability of the target data buffer.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LAPI_Get, LAPI_Put, LAPI_Putv, AM, AM_LW, AMV, AMX, GET, GETV, PUT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Completion</td>
<td>The origin task</td>
<td>LAPI_Amsend,</td>
<td>Pass it as an argument to an API call</td>
<td>Upon return of completion handler (if used), or on completion of message delivery (an internal message notifies the origin task in either case)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LAPI_Amsendv,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>LAPI_Get, LAPI_Putv, AM,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AM_LW, AMV, AMX, and PUT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Specifying target-side addresses

When working with any target-side addresses for handlers, counters, or data buffers, a value in the target task’s address space must be specified, often by the origin task. In those instances, it is important to remember that the address must be obtained from the target task. The normal mechanism for doing so is to use LAPI_Address_init or LAPI_Address_init64. If you are using the LAPI_Xfer interface for data communication, remote-side addresses are of type lapi_long_t, and must be obtained using LAPI_Address_init64.

Additional progress functions

LAPI provides functions for checking status while awaiting message delivery. Recall that progress is made in the LAPI dispatcher. When LAPI is used in polling mode (interrupts are turned off), it is possible that message packets have arrived, but the dispatcher does not get called (if nothing is being sent, for example). The user can invoke the dispatcher explicitly by calling LAPI_Probe on a LAPI handle. A call to LAPI_Probe will cause the dispatcher to check for the arrival of any message packets. See "LAPI_Probe" on page 211 for more information.
With the LAPI_Msgpoll subroutine, LAPI provides a means of running the dispatcher several times until either progress is made or a specified maximum number of dispatcher loops have executed. Here, progress is defined as the completion of either a message send operation or a message receive operation. See “LAPI_Msgpoll” on page 208 for more information.

**Bulk message transfer on AIX**

**System requirements:**
- RSCT LAPI for AIX running over US

Bulk message transfer, which uses the remote direct memory access (RDMA) protocol, is especially useful for applications that transfer relatively large amounts of data — more than 150 kilobytes (KB) — in a single call, or that overlap computation and communication, because the CPU is no longer required to copy data.

By default, bulk transfer is disabled in RSCT LAPI for AIX. To enable bulk transfer for interactive POE jobs, set the environment variable `MP_USE_BULK_XFER=yes`. Transparently to the user, this causes portions of the user’s virtual address space to be pinned and mapped to a communication adapter. LAPI then uses RDMA to move data from the send buffer to the receive buffer.

Not all communication adapters support RDMA. Bulk transfer is supported only on systems where the communication adapters’ device driver supports RDMA. To determine which systems support bulk transfer, use the TWS LoadLeveler command `lstatus` with either the `-l` or `-R` flag to display processors with adapters that support RDMA. Supporting processors have an RDMA resource listed in the command output.

To change the minimum message size for which LAPI will attempt to make bulk transfers, modify the setting of the `MP_BULK_MIN_MSG_SIZE` environment variable.

For more information about `MP_USE_BULK_XFER` and `MP_BULK_MIN_MSG_SIZE`, see “Variables for data transfer” on page 317.

These environment variables are hints that may or may not be honored by the communication library. For the communication library to honor these variables, the system administrator must:

1. Enable the RDMA protocol in the Switch Network Interface (SNI) device driver. To do this, set the `rdma_xlat_limit` attribute of the SNI devices to an appropriate value. For more information, see the appropriate SNI documentation for the specific server type, for example: *Switch Network Interface for eServer pSeries High Performance Switch: Guide and Reference*.
2. Follow the instructions in the “Using bulk data transfer” section of TWS LoadLeveler: Using and Administering.

You can use the LAPI_Qenv subroutine to find out if bulk transfer is enabled and to query the minimum message size for bulk transfer. See “LAPI_Qenv” on page 221 for more information.

The maximum message size for bulk transfer is 32 megabytes (MB). Transparently to the user, LAPI delivers messages that are larger than 32MB in 32MB chunks. The performance of bulk message transfer may be enhanced by using technical...
large pages. RDMA operations are considerably more efficient when large (16 MB) pages are used rather than small (4 KB) pages, especially for large transfers.

For more information about RDMA, see "User-initiated RDMA transfer on AIX" on page 65.

Normal LAPI message passing involves packetization of messages for transfer in sizes that can be handled by the lower-level communication subsystem. LAPI also allows for bulk transfer of messages using the adapter's direct memory access (DMA) capability. To illustrate the use of bulk transfer, let's start with an examination of the packet-mode approach to message transfer. Figure 8 illustrates the flow of LAPI packet-level message passing. Each vertical arrow represents a data copy by LAPI through calls to the communication subsystem layer. There is one send-side copy into a network FIFO from a user data buffer and one receive-side copy out of a network FIFO into a user data buffer.

For larger messages, the AIX communication subsystem (CSS) supports a DMA method on both sides of the communication, resulting in one less data copy on each of the sending and receiving tasks. Figure 9 on page 65 illustrates the flow of data for a LAPI bulk transfer. Using a rendezvous protocol, the origin and target tasks establish DMA connections with the adapter firmware. The target task then pulls the data across the switch. Using the DMA method, the switch adapter transfers the data from the origin task's address space directly into the target task's address space. If any step in the sequence fails, LAPI will use packet mode for message delivery.
User-initiated RDMA transfer on AIX

**System requirements:**
The HPS with RSCT LAPI for AIX

Using LAPI’s remote direct memory access (RDMA) interface, users can put data in or get data from a remote task’s address space, with no CPU usage at the remote task. Optionally, the remote task can receive notification that an RDMA operation has completed on its address space. RDMA read and write operations are initiated by a call to LAPI_Xfer with transfer type LAPI_RDMA_XFER (RDMA) specified. The corresponding structure passed into this call, lapi_hwxfer_t, contains the information that is required to initiate the transfer. See [lapi_hwxfer_t details](page 271) for more information.

**Resources required for an RDMA job**

At a minimum, an RDMA operation requires a mechanism for the adapter to directly access memory in the user’s address space on the initiator and target address spaces, as well as adapter resources at the RDMA target. The addresses involved in the RDMA transfer must be first registered using a LAPI_Util function call (see [RDMA setup](page 66)). The handle returned by the registration call is called a protocol virtual offset (PVO). The source and target addresses for an RDMA operation are each specified by a PVO and an offset from the beginning of the corresponding registered address space.

In addition to the local and remote memory specification, a remote RDMA context (rCxt) is required for the target task. The rCxt specifies remote adapter resources that are required to perform the RDMA operation. When multiple adapters are used for striped RDMA communication, the remote rCxt encapsulates the adapter resources required among all of the adapters that are assigned to the remote task.

From a user perspective, the PVO and the remote rCxt are opaque objects.
RDMA setup

Before starting an RDMA operation, the user must obtain local and remote PVOs and a remote rCxt. Optionally, a counter update or callback can be registered when target notification is required.

A PVO is obtained by specifying an address range using the `LAPI_Util` subroutine. The address range is specified by an address and length in a structure that is passed to `LAPI_Util`. In order for this operation to succeed, the user must have read/write access to the entire specified address range. The `LAPI_Util` operation pins the corresponding user memory and returns the PVO as a handle to access this memory. This operation is entirely a local operation and the PVO pertains only to the local address space where it was obtained. A remote task can use this PVO as a target PVO. However, this requires separate non-RDMA communication to transfer the PVO to the remote task for it to initiate an RDMA operation to the address range corresponding to the PVO. For example, this transfer of PVOs can be done with an active message (AM) transfer or with a call to `LAPI_Address_init64`. The same `LAPI_Util` call with a different option can be used to release the PVO in order to reclaim pinned memory resources.

The remote rCxt used for RDMA operations is also obtained with a call to `LAPI_Util`. In this case, a target task is specified. This rCxt can be used only for RDMA operations to the target task specified in the `LAPI_Util` call. The call to obtain the remote rCxt is a blocking call that communicates with the target task. The same `LAPI_Util` call with a different option can be used to release the remote rCxt in order to reclaim adapter RDMA resources.

A final `LAPI_Util` call allows the user to specify a callback function, a counter update, or both when an RDMA operation completes on the target side. This registration function call associates a 16-bit tag with a callback function (and associated pointer) or counter. The initiator of an RDMA operation can specify the same tag and set a flag so that the target side is notified through a callback or counter update. If interrupts are disabled, this target side notification occurs only after a call to a LAPI probe or data transfer function.

RDMA striping

When multiple user space windows are requested for communication striping (see Chapter 15, "Striping, failover, and recovery," on page 127 for details on how this is done), the LAPI protocol transparently manages the distribution of RDMA communication among the multiple adapter windows. Such a distribution of RDMA communication among multiple adapters can result in significant improvement in communication bandwidth. The LAPI protocol also manages failover and recovery of RDMA communication when one of the adapters involved in the communication fails or recovers.

RDMA considerations

When using the RDMA interface, consider the following:

- The first RDMA operation could be slow because extra rendezvous calls might be needed to complete some internal LAPI setup.
- The DMA engine is most efficient when data is transferred on a cache line boundary. If possible, initiate RDMA operations on even 128-byte boundaries on the source side and the target side.
- There is a limit to the amount of memory and the number of memory registrations that can be done. This limit is influenced by the amount of real
memory, number of large pages, and the translation limit that is set in the device driver. It is preferable to register a few large memory regions, rather than many small ones.

- If each task has access to a shared memory region, only a single task needs to register that memory region. The resulting PVO can be used as a source or target PVO for any task accessing that shared memory.
- Counter updates and callbacks occur only when data has been transferred completely.
- There are a total of 798 blocks of rCxts for the HPS adapter. Each block of rCxts contains 128 rCxts.
- Limits on the number of rCxts could prevent RDMA from being used for very large scaling. For example, when running with 64 tasks per adapter and one rCxt per destination, a job is limited to 1024 tasks, as shown in Table 19 and Table 20.

For information about the registration functions and RDMA transfer functions, see “LAPI_Util” on page 241 and “LAPI_Xfer” on page 260.

Table 19 shows how many blocks of rCxts are required to communicate with every task in the job, for each window instance. For each extra rCxt per destination, these values must be doubled.

**Table 19. Blocks of rCxts per task that are required for each window for an RDMA job**

<table>
<thead>
<tr>
<th>Number of tasks</th>
<th>One rCxt per destination</th>
<th>Two rCxts per destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>64</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>256</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>1024</td>
<td>9</td>
<td>17</td>
</tr>
<tr>
<td>4096</td>
<td>33</td>
<td>65</td>
</tr>
<tr>
<td>8192</td>
<td>65</td>
<td>129</td>
</tr>
</tbody>
</table>

**Table 20. Total blocks of rCxts per adapter required with multiple tasks sharing an adapter**

<table>
<thead>
<tr>
<th>Number of tasks</th>
<th>64 tasks per adapter</th>
<th>32 tasks per adapter</th>
<th>16 tasks per adapter</th>
<th>8 tasks per adapter</th>
<th>4 tasks per adapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>not applicable</td>
<td>not applicable</td>
<td>not applicable</td>
<td>not applicable</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>not applicable</td>
<td>not applicable</td>
<td>32</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>64</td>
<td>not applicable</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>256</td>
<td>192</td>
<td>96</td>
<td>48</td>
<td>24</td>
<td>12</td>
</tr>
<tr>
<td>1024</td>
<td>576</td>
<td>288</td>
<td>144</td>
<td>72</td>
<td>36</td>
</tr>
<tr>
<td>4096</td>
<td>not enough adapter resources</td>
<td>not enough adapter resources</td>
<td>528</td>
<td>264</td>
<td>132</td>
</tr>
<tr>
<td>8192</td>
<td>not enough adapter resources</td>
<td>not enough adapter resources</td>
<td>not enough adapter resources</td>
<td>520</td>
<td>260</td>
</tr>
</tbody>
</table>

**A sample RDMA program**

The following example shows how to create a simple RDMA get/put function.

```c
#include <stdlib.h>
#include <stdio.h>
```
#include <string.h>
#include <unistd.h>
#include <lapi.h>
/* For any error messages returned by LAPI */
char err_msg_buf[LAPI_MAX_ERR_STRING];

/* Constant for array lengths */
#define ARRAYLEN 10

/** Macro to check return code of function calls. Keeps return 
** code checking logic from needing to be in main logic. */
#define CHECK(func_and_args) 
{ 
  int rc;
  if ((rc = (func_and_args)) != LAPI_SUCCESS) {
    LAPI_Msg_string(rc, err_msg_buf);
    fprintf(stderr, 
            
            (func_and_args) returns error: %d

            , rc);
    exit(1);
  }
}

/** Sample Program to illustrate the use of the LAPI_Xfer interface 
** to the hardware get/put routine using the RDMA interface. 
** For a set of n tasks 0,1,...,n-1, where n is an even number, all 
** tasks are divided into (src,tgt) buddy pairs (0,1), (2,3), etc. 
** For each pair, an array of ints is transferred from src to tgt 
** and back with LAPI_Xfer calls. The target counter is used for 
** synchronization. Once the message completes, the target shows 
** the result by printing the contents of the data buffer. */

/* Stores initial value on source and final value on target */
int data_buffer[ARRAYLEN];
/* Stores list of remote PVOs */
lapi_user_pvo_t *pvo_list; /* remote data buffer addr */
/* Local PVO of data buffer */
lapi_user_pvo_t pvo;
/* Updates on target at message completion */
lapi_cntr_t tgt_cntr;
/* Updates on origin at message completion */
lapi_cntr_t org_cntr;

int main( int argc, char **argv )
{
  lapi_handle_t handle;  /* LAPI handle */
lapi_info_t info;  /* Info to pass to LAPI_Init */
int task_id;  /* Our LAPI task ID */
int num_tasks;  /* Total number of tasks */
int i,j;  /* Loop counters */
int buddy;  /* Our communication partner */
int val;  /* Needed for waitcntr call on source */
lapi_xfer_t xfer_struct;  /* Data structure for the Xfer call */
lapi_get_pvo_t util_pvo;  /* For call to obtain PVO */
lapi_rdma_tag_t lapi_rdma_tag;  /* RDMA notification tag */
lapi_remote_cxt_t util_cxt;  /* For call to obtain rCxt */
lapi_rdma_notification_t util_cntr; /* For call to register counter */
lapi_user_cxt_t remote_cxt; /* Remote context for RDMA call */

/* Not passing any info to init through this structure */
bzero(&info, sizeof(lapi_info_t));

/* Initialize the LAPI handle */
CHECK((LAPI_Init(&handle, &info)));

/* Query LAPI for our task ID */
CHECK((LAPI_Qenv(handle, TASK_ID, &task_id)));

/* Query LAPI for the total number of tasks in the job */
CHECK((LAPI_Qenv(handle, NUM_TASKS, &num_tasks)));

/* This example only supports even numbers of tasks */
if ( (num_tasks % 2) != 0 || (num_tasks < 2) ) {
    fprintf(stderr,"Error: this program uses an even number of tasks, but was called with %d\n", num_tasks);
    exit(1);
}

pvo_list = (lapi_user_pvo_t *) malloc(num_tasks*sizeof(lapi_user_pvo_t));

/* Set target counter used on task 0 */
CHECK((LAPI_Setcntr(handle,&tgt_cntr,0)));

/* Set origin counter used on task 1 */
CHECK((LAPI_Setcntr(handle,&org_cntr,0)));

/* Translate and map the buffer to the adapter */
util_pvo.Util_type = LAPI_XLATE_ADDRESS;
util_pvo.length = ARRAYLEN*sizeof(int);
util_pvo.usr_pvo = 0;
util_pvo.address = data_buffer;
util_pvo.operation = LAPI_RDMA_ACQUIRE;
bzero(&data_buffer, sizeof(int)*ARRAYLEN);
CHECK(LAPI_Util(handle, (lapi_util_t *) &util_pvo));
pvo = util_pvo.usr_pvo;

/* Collective calls. Each task stores its own address plus remote
** addresses. */
CHECK((LAPI_Address_init64(handle,(lapi_long_t)pvo,pvo_list)));
lapi_rdma_tag = 22; /* Can be any number that fits in ushort */

if ( task_id % 2 == 0 ) { /* Target of RDMA operation */
    /* Register the target counter for RDMA notification */
    util_cntr.Util_type = LAPI_REGISTER_NOTIFICATION;
    util_cntr.rdma_tag = lapi_rdma_tag;
    util_cntr.flags = LAPI_RCNTR_UPDATE;
    util_cntr.cntr = &tgt_cntr;
    util_cntr.callback = NULL;
    CHECK(LAPI_Util(handle, (lapi_util_t *) &util_cntr));

    /* Collective call. Synchronize before starting data transfer. */
    CHECK((LAPI_Gfence(handle)));
    /* ** Up to this point, all instructions have executed on all tasks.
** We now begin differentiating tasks. */
    buddy = task_id + 1;
    /* Wait to obtain data from data source */
    CHECK((LAPI_Waitcntr(handle, &tgt_cntr,2, &val)));
    printf("Received data from buddy %d\n", buddy);
    for (i = 0; i < ARRAYLEN; i++) {
        printf("data_buffer[%d]: %d\n", i, data_buffer[i]);
    }
} else {  /* Initiator of RDMA operation */
    buddy = task_id - 1;
    CHECK((LAPI_Gfence(handle)));

    for (i = 0; i < ARRAYLEN; i++) {
        data_buffer[i] = i*(1+task_id);
    }

    /* Obtain remote rCxt for RDMA operation */
    util_cxt.Util_type = LAPI_REMOTE_RCXT;
    util_cxt.operation = LAPI_RDMA_ACQUIRE;
    util_cxt.dest = buddy;
    CHECK((LAPI_Util(handle, (lapi_util_t *) &util_cxt));
    remote_cxt = util_cxt.usr_rcxt;

    bzero(&xfer_struct, sizeof(xfer_struct));
xfer_struct.Hwxfer.Xfer_type = LAPI_RDMA_XFER;
xfer_struct.Hwxfer.tgt = buddy;
xfer_struct.Hwxfer.op = LAPI_RDMA_PUT|LAPI_RCNTR_UPDATE;
xfer_struct.Hwxfer.rdma_tag = lapi_rdma_tag;
xfer_struct.Hwxfer.remote_cxt = remote_cxt;
xfer_struct.Hwxfer.src_pvo = pvo;
xfer_struct.Hwxfer.tgt_pvo = pvo_list[buddy];
xfer_struct.Hwxfer.src_offset = 0;
xfer_struct.Hwxfer.tgt_offset = 0;
xfer_struct.Hwxfer.len = (ulong) ARRAYLEN*(sizeof(int));
xfer_struct.Hwxfer.shdlr = (scompl_hndlr_t *) NULL;
xfer_struct.Hwxfer.sinfo = (void *) NULL;
xfer_struct.Hwxfer.org_cntr = &org_cntr;
    CHECK((LAPI_Xfer(handle, &xfer_struct)));

    /* Wait for RDMA completion */
    CHECK((LAPI_Waitcntr(handle, &org_cntr, 1, &val)));

    /* Set data back to 0 */
    for (i = 0; i < ARRAYLEN; i++) {
        data_buffer[i] = 0;
    }

    bzero(&xfer_struct, sizeof(xfer_struct));
xfer_struct.Hwxfer.Xfer_type = LAPI_RDMA_XFER;
xfer_struct.Hwxfer.tgt = buddy;
xfer_struct.Hwxfer.op = LAPI_RDMA_GET|LAPI_RCNTR_UPDATE;
xfer_struct.Hwxfer.rdma_tag = lapi_rdma_tag;
xfer_struct.Hwxfer.remote_cxt = remote_cxt;
xfer_struct.Hwxfer.src_pvo = pvo;
xfer_struct.Hwxfer.tgt_pvo = pvo_list[buddy];
xfer_struct.Hwxfer.src_offset = 0;
xfer_struct.Hwxfer.tgt_offset = 0;
xfer_struct.Hwxfer.len = (ulong) ARRAYLEN*(sizeof(int));
xfer_struct.Hwxfer.shdlr = (scompl_hndlr_t *) NULL;
xfer_struct.Hwxfer.sinfo = (void *) NULL;
xfer_struct.Hwxfer.org_cntr = &org_cntr;
    CHECK((LAPI_Xfer(handle, &xfer_struct)));

    /* Wait for RDMA completion */
    CHECK((LAPI_Waitcntr(handle, &org_cntr, 1, &val)));

    printf("Received data from buddy %d\n", buddy);
    for (i = 0; i < ARRAYLEN; i++) {
        printf("data_buffer[%d]: %d\n", i, data_buffer[i]);
    }

    util_cxt.Util_type = LAPI_REMOTE_RCXT;
    util_cxt.operation = LAPI_RDMA_RELEASE;
Using RDMA with the InfiniBand switch

RSCT LAPI for AIX supports RDMA communication over the InfiniBand switch. To provide this support, LAPI creates reliable connected queue pairs (RC QPs) for a pair of tasks to establish adapter resources, based on the values you provide using various environment variables. These RC QPs can be used for RDMA communication between this pair of tasks for all future contiguous messages that are larger than the RDMA threshold. Use the `MP_BULK_MIN_MSG_SIZE` environment variable to determine the RDMA threshold. You can also specify a maximum number of RC QPs that can be created for a task by setting the `MP_RC_MAX_QP` environment variable. Limiting the number of RC QPs per task allows you to reserve memory that can be used for computation. The RC QPs exist until the job is terminated, checkpointed, or preempted. LAPI uses the following environment variables to create the RC QPs:

**MP_RC_MAX_QP**

- Specifies the maximum number of RC QPs that can be created. The allowable value is any positive integer. The default is **8156**.
- `MP_RC_MAX_QP` limits the amount of memory that is consumed by RC QPs. It is suggested that you only set this variable if you suspect that your application is performing poorly due to lack of memory.

**MP_RC_USE_LMC**

- Determines whether LID Mask Control (LMC) is enabled. Enabling the use of LMC can improve performance, because a single port can support multiple RC paths. The default value is **no** (only one RC connected path is supported). Setting `MP_RC_USE_LMC` to **yes** causes multiple RC paths to be supported, which may improve performance.

To learn more about InfiniBand and RC QPs, go to:

[http://www.infinibandta.org/specs](http://www.infinibandta.org/specs)
Chapter 8. Active messaging

The active message function (LAPI_Amsend) is a non-blocking call that causes the specified active message handler to be invoked and executed in the address space of the target process. Completion of the operation is signaled if counters are specified. Both standard and synchronous behaviors are supported. The LAPI_Amsend subroutine provides two counters: the origin counter (org_cntr) and the completion counter (cmpl_cntr), which can be used to provide the two behaviors. With standard behavior, LAPI increments the origin counter (org_cntr) when the origin buffer can be reused. With synchronous behavior, LAPI increments the completion counter (cmpl_cntr) after the completion handler has completed execution.

The LAPI_Xfer subroutine provides functions that are similar to LAPI_Amsend, with some enhancements. See "LAPI_Xfer" on page 260 for more information.

Flow of active message operations

Figure 10 on page 74 illustrates the sequence of events for a LAPI_Amsend operation or a LAPI_Amsendv operation. Because they use a header handler and a completion handler, active message operations are more complex than transfer operations. Upon arrival of the first packet, the user header handler is called. This handler is always run inline. For this reason, it is important to keep the body of the header handler small so that progress on other messages will not be blocked for long. The header handler returns a data buffer address to LAPI, which writes the data starting at that address. Once the final packet arrives, LAPI increments the target counter and either directly invokes or enqueues the completion handler for later execution in a completion handler thread. See "The completion handler" on page 58 for more information.
Using LAPI_Amsend: a complete LAPI program

This section contains a complete listing of a LAPI program. A line-by-line description of the program follows this listing. The source code shown demonstrates a single LAPI_Amsend call, complete with header handler and completion handler definitions. This listing is a modified version of the sample program Am.c, which is available in the LAPI samples directory (/opt/rsct/lapi/samples/lapi_api). For more information, see Chapter 22, “Sample LAPI programs,” on page 291.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <lapi.h>

/* for any error messages returned by LAPI */
char err_msg_buf[LAPI_MAX_ERR_STRING];

/* constant for array lengths */
define ARRAYLEN 10

/* list of header handler addresses */
void **hdr_hdlr_list;

/* stores initial value on src and final value on tgt */
int data_buffer[ARRAYLEN];

/* updates on src at msg completion */
lapi_cntr_t compl_cntr;

/* completion handler that runs on the target after completion */
```
** of message delivery. prints the contents of the data buffer that is the target of the LAPI_Amsend call. */

```c
void compl_hndlr(lapi_handle_t *hndl, void *completion_param)
{
  int i; /* loop counter */

  printf("Greetings from the completion handler...
");  
  for( i = 0; i < ARRAYLEN; i++ ) {
    printf("final buffer[\%d\]: %d
", i, data_buffer[i]);
  }
}
```

/* header handler that runs on target when first packet arrives. 
** sets the completion_handler pointer and returns the address 
** of the data buffer for message delivery. LAPI writes the 
** Amsend data at this address. */

```c
void *header_handler(lapi_handle_t *hndl, void *uhdr, uint *uhdr_len,
  ulong *msg_len, compl_hndlr_t **comp_h,
  void **info) {
  /* pass the completion handler back by reference */
  *comp_h = compl_hndlr;
  return data_buffer;
}
```

```c
int main( int argc, char **argv )
{
  lapi_handle_t hndl; /* LAPI handle */
  lapi_info_t info; /* Info to pass to */
  /* LAPI_Init */
  int task_id; /* Our LAPI task ID */
  int num_tasks; /* Total number of tasks */
  int i; /* Loop counter */
  int val; /* Needed for waitcntr call */
  int buddy; /* Our communication */

  /* Clear the structure. Not passing any information 
  ** to initialize through this structure. */
  bzero(&info, sizeof(lapi_info_t));

  /* Initialize the LAPI handle */
  LAPI_Init(&hndl, &info);

  /* Query LAPI for our task ID */
  LAPI_Qenv(hndl, TASK_ID, &task_id);

  /* Query LAPI for the total number of tasks in the job */
  LAPI_Qenv(hndl, NUM_TASKS, &num_tasks);

  /* This example only supports even numbers of tasks */
  if ( (num_tasks < 2) || ((num_tasks % 2) != 0) ) {
    fprintf(stderr,"ERROR: this example requires an even number of tasks, but has been invoked with
    " "\%d\", num_tasks);
    exit(1);
  }

  /* get address of header handler on target */
```
include standard header files for LAPI programs.

Lines 7 and 8 declare a string buffer for retrieving messages that are associated
with non-zero LAPI return codes. Calling `LAPI_Msg_string(rc, err_msg_buf)` will store the message associated with `rc` in `err_msg_buf`.

**Line 11** defines a fixed data length for this example.

**Line 14** declares a pointer to maintain a list of header handler addresses for each task. The header handler runs on the target task, so the address that is specified for the header handler pointer must be in the target address space.

**Line 17** declares a simple data buffer. Because it is declared in common code, each task will have this buffer allocated in its address space. This program initializes a set of values in the sender task to this buffer, clears this set of values in the receiver task, then transfers the original set of values from the sender to the receiver.

**Line 20** specifies the only LAPI counter that this program will require for synchronization.

**Lines 22 to 36** specifies the completion handler that runs at the target upon completion of message delivery. When executed, this guarantees that all data from the transfer has been written to the desired target address. In this example, the program simply prints the results. In practice, completion handlers can be used for many things, including additional data manipulation or synchronization (such as by incrementing counters that the user is managing internally). Completion handlers are not required for active messaging calls. If no completion handler is desired, simply pass NULL to the completion handler pointer in the user header handler.

**Lines 38 to 51** specifies the user header handler that LAPI will call when the first packet arrives at the target. At a minimum, it must return to LAPI the buffer address at which data is to be written. It can also return (by reference parameters) a pointer to the user’s completion handler and a pointer to a structure containing any information to pass to the completion handler when it is invoked.

Header handlers are always run inline. That is, progress on any other messages is blocked until the header handler completes. For this reason it is advised that the user keep execution time in the header handler to a minimum.

**Line 54** Main begins here. This example is short: only one LAPI communication operation is done before the program exits. For this reason, it is not a problem to keep all of the program flow in main. Recall that a separate application of this code will be executed for each task. As such, all code will execute on all tasks. This program branches on task ID at line 130 to ensure that certain code only executes on one set of tasks, and the other branch executes only on the other set of tasks. After the branched sections, control rejoins and the remaining code is executed on all tasks. It is helpful to think of this fork/join model as you write LAPI programs.

**Lines 57 to 65** include variable declarations. The handle is the user’s abstraction of a LAPI instance. After initialization (line 74 in this case), the handle is used to interact with the LAPI subsystem. Users can use the
handle to query runtime parameters and execute data transfers. As part of cleanup, the handle should be properly terminated (line 150).

The lapi_info_t structure is used to pass certain information to LAPI_Init. Most often, it is used for passing a pointer to the user’s error handler. Certain fields in the structure are not used; LAPI will return an error if these fields contain values. For this reason, it is important to clear the structure (line 71), and then set any desired values. This example does not use any of the fields, so the program leaves the memory cleared. See "Registering an error handler" on page 43 for an example of using the lapi_info_t structure to register a user error handler.

LAPI allows us to query our task ID. At line 99, the code stream splits. One branch will execute on all tasks with even-numbered IDs (task_id %2 == 0), and the other branch will execute on tasks with odd IDs.

In this example, task IDs are used to set up buddy pairs for data transfer. Buddy pairs are established as (0,1), (2,3), and so on, such that the lower task ID in the pair is the even number. The lower (even) task ID drives the communication by sending a message to the odd-numbered task.

The buddy-pair concept is a specific case of task grouping, where one task is the "master" for a group of tasks. The "master task" drives all communication and synchronization. It is possible to have master tasks for groups of larger cardinalities. One common model is to use task 0 as a master task and all other tasks as "slave tasks".

Similar to task_id, LAPI has a mechanism for querying the total number of tasks in the job. Because this example communicates in task pairs, it requires an even number of tasks. This program uses the value queried from LAPI to check the number of tasks on line 83 and exits if this value is not even.

The remaining, basic variables are needed for various tasks.

**Line 71**
clears the fields in the lapi_info_t structure before passing it to LAPI. See the description of lines 57 to 65 for information about lapi_info_t.

**Line 74**
initializes the LAPI handle. This call creates a LAPI instance and initializes the communication subsystem. After this call completes, any query and communication API calls can be performed on this handle, until this handle is terminated.

**Lines 76 to 80**
Lines 77 and 80 are executed in most programs that use LAPI, because the task ID and the total number of tasks in the job are commonly-used values. See the description of lines 57 to 65 for more information on how these values are used in this example.

**Lines 82 to 88**
check to make sure that the number of tasks is even. See the description of lines 57 to 65 for more information on task topology.

**Lines 90 to 93**
Recall that the header handler executes on the target, but the
address of the function must be known to the source task. This program uses the `LAPI_Address_init` subroutine to exchange header handler address data among all of the tasks. `LAPI_Address_init` expects a buffer with enough space to hold an address for every task in the job, so this program does the required `malloc` first at line 91, then frees it at line 147. Note that since we are still in common code, the `malloc` and `LAPI_Address_init` call will be executed at this point on every task, which is what we want.

**Line 99**  
Control forks here. This branch will only be entered by even-numbered tasks (the "drivers" in this example). The branch for odd-numbered tasks begins at line 130.

**Line 102**  
Because this task will drive, it needs to keep track of its buddy's task ID.

**Line 104**  
populates the buffer with data to be transferred. The values are relative to the ID of the buddy task so they can be easily and uniquely verified after being received by the buddy.

**Lines 109 to 113**  
LAPI communication API calls are non-blocking; message delivery is asynchronous. LAPI provides counters that signify different events. In this example, we want to know when message delivery at our buddy is complete. For this purpose, we use a local completion counter, which LAPI will increment once the completion handler on the target returns. We will wait on this counter in a `LAPI_Waitcnt` call at line 128, immediately after the `LAPI_Amsend` call on line 122 returns.

**Line 116**  
This is a collective data fence operation. All calls that are made before this point have no asynchronous side effects and thus are guaranteed to complete before we leave this fence. Our buddy has a matching `LAPI_Gfence` call at line 140.

**Lines 118 to 125**  
This is the actual communication API call. As above, all LAPI calls are made with respect to the handle returned from `LAPI_Init`. We are sending data to our buddy task. For the header handler pointer, we pass the entry returned from the collective `LAPI_Address_init` call on line 92. The address passed into the call by our buddy task is stored at index buddy in the table.

We are not using a user header in this example, so the next two arguments (the user header and user header length) are NULL and 0, respectively. See the sample program `vector/accumulate_and_return.Amv.c` for an example of using a user header.

The next argument is the base data address from which data is to be read (`&(data_buffer[0])`) followed by the number of bytes to transfer (`ARRAYLEN*(sizeof(int))`).

The final three arguments are pointers to counters that LAPI will manage. As mentioned above, we are only concerned in this case with the completion counter. So we pass NULL for both the target and origin counters. The target counter will increment on the target when message delivery is complete, so if used, requires a remote address to be passed. See the sample program `xfer/Put_xfer.c` for an example of using a target counter.
When used, the origin counter increments locally when the origin data buffer is no longer needed by LAPI and can thus be reused by the origin task. See the `vector/strided.c` example for demonstration of origin counter usage.

As mentioned above, we are using the completion counter in this case and so pass its address.

**Line 128**
We wait here for LAPI to signal message completion. This completes the branch specific to even-numbered tasks.

**Line 130**
This begins the branch that will only be executed by odd-numbered (receiver) tasks.

**Line 131**
Our buddy is driving the communication, so we do not need its ID. This comment is just left in as a reminder.

**Lines 135 to 137**
Similarly, we zero out the data space here so that we know that the data in the final output must have come from a remote task.

**Line 140**
This call matches the sender’s `LAPI_Gfence` call on line 116.

**Line 141**
Code paths re-join here. All remaining code is executed on both tasks.

**Lines 143 to 150**
represent a standard LAPI shutdown sequence. All tasks execute a final `LAPI_Gfence` call (line 144), then clean up any allocated data structures (line 147). Any frees should be done before executing `LAPI_Term`.

**Line 152**
Finally, we return success. If POE is running, it will pass this code back to the operating system for the process return code.
Chapter 9. Collecting statistics and querying resources

You can use the LAPI_Qenv subroutine to:

- print communication statistics
- query communication statistics
- query network resources

See “LAPI_Qenv” on page 221 for more information.

You can use the pnsd_stat command to query the statistics of user space tasks.

You can use the pnsd_trigger command to call a trigger.

Printing data transfer statistics

When passed the PRINT_STATISTICS query type, LAPI_Qenv sends data transfer statistics to standard output. In this case, ret_val is unaffected. However, LAPI's error checking requires that the value of ret_val is not Null (in C) or LAPI_ADDR_NULL (in FORTRAN) for all LAPI_Qenv types (including PRINT_STATISTICS).

This query type supports multiple parallel APIs. LAPI maintains separate communication statistics for each parallel API.

Querying UDP/IP statistics

When passed the QUERY_STATISTICS query type, LAPI_Qenv interprets ret_val as a pointer to type lapi_statistics_t. Upon function return, the fields of the structure contain LAPI’s data transfer statistics for data transferred using UDP/IP. QUERY_STATISTICS expects a pointer to type lapi_statistics_t. The address of stats is cast to int *, which is required to match the signature of LAPI_Qenv. For example:

```c
lapi_statistics_t stats;
LAPI_Qenv(handle, QUERY_STATISTICS, (int *)&stats));
```

This query type supports multiple parallel APIs. LAPI maintains separate communication statistics for each parallel API.

Querying user space statistics

When passed the QUERY_STATISTICS query type, LAPI_Qenv interprets ret_val as a pointer to type lapi_statistics_t. Upon function return, the fields of the structure contain LAPI’s data transfer statistics for data transferred using the user space (US) protocol. QUERY_STATISTICS expects a pointer to type lapi_statistics_t. The address of stats is cast to int *, which is required to match the signature of LAPI_Qenv. For example:

```c
lapi_statistics_t stats;
LAPI_Qenv(handle, QUERY_STATISTICS, (int *)&stats));
```

This query type supports multiple parallel APIs. LAPI maintains separate communication statistics for each parallel API.
For US tasks

Use the `pnsd_stat` command to query the statistics of user space tasks. When `pnsd_stat` is run on a computation node, it gives only the statistics of the node. See "pnsd_stat" on page 148 for more information.

Calling a trigger

Use the `pnsd_trigger` command to call a trigger. See "pnsd_trigger" on page 150 for more information.

Querying local send statistics

When passed the `QUERY_LOCAL_SEND_STATISTICS` query type, `LAPI_Qenv` interprets `ret_val` as a pointer to type `lapi_statistics_t`. Upon function return, the fields of the structure contain LAPI's data transfer statistics for data transferred through intra-task local copy. For example:

```c
lapi_statistics_t stats;
LAPI_Qenv(handle, QUERY_LOCAL_SEND_STATISTICS, (int *)&stats));
```

With this query, you can obtain the byte count of the data that is transferred through the local copy path when the source and target are the same. The following statistics are reported: the number of bytes sent and the number of bytes received for the local copy path, the number of packets sent, and the number of packets received. The number of bytes sent for a message are counted only for data messages that are sent successfully. The number of bytes received for a message are counted only for data messages that complete successfully. As with adapter statistics, LAPI also provides separate sets of statistics for LAPI-only traffic and shared traffic in the local copy path. The packet count reported for the local copy path will always be 0 because data transfer in this path does not use packetization.

This query type supports multiple parallel APIs. LAPI maintains separate communication statistics for each parallel API.

Querying shared memory statistics

When passed the `QUERY_SHM_STATISTICS` query type, `LAPI_Qenv` interprets `ret_val` as a pointer to type `lapi_statistics_t`. Upon function return, the fields of the structure contain LAPI's data transfer statistics for data transferred through shared memory. `QUERY_SHM_STATISTICS` expects a pointer to type `lapi_statistics_t`. For example:

```c
lapi_statistics_t stats;
LAPI_Qenv(handle, QUERY_SHM_STATISTICS, (int *)&stats));
```

With this query, you can obtain the byte count of the data that is transferred through the shared memory path when the source and the target reside on the same node. The following statistics are reported: the number of bytes sent and the number of bytes received for the shared memory path, the number of packets sent, the number of packets received, and the number of retransmit packets. A packet in shared memory path refers to a shared memory slot when using slot transfer or a shared memory attachment when using attach transfer. The number of retransmit packets in the shared memory path refers to the number of times when attaching a shared memory segment fails. The number of bytes sent via shared memory is
defined as all the data bytes sent through shared memory either by slot path or attach path. The number of bytes sent for a message are counted only for data messages that are sent successfully. If attaching a shared memory segment does not succeed for transferring a message and that message has been failed over to the slot path, only slot path data bytes are counted into the number of bytes sent. The number of bytes received for a message are counted only for data messages that complete successfully. As with adapter statistics, LAPI also provides separate sets of statistics for LAPI-only traffic and shared traffic in the shared memory path.

This query type supports multiple parallel APIs. LAPI maintains separate communication statistics for each parallel API.

See "LAPI shared memory" on page 306 for more information.

### Querying network resources

With the NETWORK_RESOURCES query type, users (MPI in particular) can get replacements for network strings. The user performing the query needs to pass in the address of a char pointer. Upon return, the address holds a pointer to a char string in LAPI. The user should not modify the char string. The returned string is similar in format to the network string value of certain environment variables.

In IP mode, this format is:

@number-of-stripes;IP-address,interface; ... ;IP-address,interface

For example:

@2;192.168.107,ib0;192.168.9.107,ib1

IPv4 and IPv6 addresses are supported.

In user space mode, this format is:

@number-of-stripes;window,adapter; ... ;window,adapter

For example:

@2;0,iba0;64;iba1
Chapter 10. Using LAPI’s profiling interface

LAPI’s profiling interface includes wrappers for each LAPI function, so you can collect data about each of the LAPI calls. For example, you can write a program that records the message size that is used in each call. This interface supports applications that are written in C, C++, and FORTRAN.

Each LAPI subroutine has a “name-shifted” interface. Suppose your program calls a LAPI subroutine, such as LAPI_Get. When you link your program with the LAPI library, the LAPI function call results in a call to the corresponding name-shifted LAPI subroutine, in this case PLAPI_Get, using the same parameters and returning the same function result. You can create a profiling library that contains function implementations for each LAPI subroutine that it will override. When you link your program with a profiling library, this library calls the corresponding name-shifted LAPI subroutine without modifying any parameters and returns the same function result. When LAPI function calls are profiled, a wrapper (for each LAPI function to be profiled) collects the profiling data and calls the name-shifted LAPI subroutine exactly as if it was called by your program.

Table 21 lists LAPI’s profiling interfaces.

<table>
<thead>
<tr>
<th>LAPI subroutine</th>
<th>C and C++ profiling interface</th>
<th>FORTRAN profiling interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPI_Addr_get</td>
<td>PLAPI_Addr_get</td>
<td>plapi_addr_get, plapi_addr_get_, PLAPI_ADDR_GET, PLAPI_ADDR_GET_</td>
</tr>
<tr>
<td>LAPI_Addr_set</td>
<td>PLAPI_Addr_set</td>
<td>plapi_addr_set, plapi_addr_set_, PLAPI_ADDR_SET, PLAPI_ADDR_SET_</td>
</tr>
<tr>
<td>LAPI_Address</td>
<td>PLAPI_Address</td>
<td>plapi_address, plapi_address_, PLAPI_ADDRESS, PLAPI_ADDRESS_</td>
</tr>
<tr>
<td>LAPI_Address_init</td>
<td>PLAPI_Address_init</td>
<td>plapi_address_init, plapi_address_init_, PLAPI_ADDRESS_INIT, PLAPI_ADDRESS_INIT_</td>
</tr>
<tr>
<td>LAPI_Address_init64</td>
<td>PLAPI_Address_init64</td>
<td>plapi_address_init64, plapi_address_init64_, PLAPI_ADDRESS_INIT64, PLAPI_ADDRESS_INIT64_</td>
</tr>
<tr>
<td>LAPI_Amsend</td>
<td>PLAPI_Amsend</td>
<td>plapi_amsend, plapi_amsend_, PLAPI_AMSEND, PLAPI_AMSEND_</td>
</tr>
<tr>
<td>LAPI_Amsendv</td>
<td>PLAPI_Amsendv</td>
<td>plapi_amsendv, plapi_amsendv_, PLAPI_AMSENDV, PLAPI_AMSENDV_</td>
</tr>
<tr>
<td>LAPI_Fence</td>
<td>PLAPI_Fence</td>
<td>plapi_fence, plapi_fence_, PLAPI_FENCE, PLAPI_FENCE_</td>
</tr>
<tr>
<td>LAPI_Get</td>
<td>PLAPI_Get</td>
<td>plapi_get, plapi_get_, PLAPI_GET, PLAPI_GET_</td>
</tr>
<tr>
<td>LAPI_Getcntr</td>
<td>PLAPI_Getcntr</td>
<td>plapi_getcntr, plapi_getcntr_, PLAPI_GETCNTR, PLAPI_GETCNTR_</td>
</tr>
<tr>
<td>LAPI_Getv</td>
<td>PLAPI_Getv</td>
<td>plapi_getv, plapi_getv_, PLAPI_GETV, PLAPI_GETV_</td>
</tr>
<tr>
<td>LAPI_Gfence</td>
<td>PLAPI_Gfence</td>
<td>plapi_gfence, plapi_gfence_, PLAPI_GFENCE, PLAPI_GFENCE_</td>
</tr>
<tr>
<td>LAPI_Init</td>
<td>PLAPI_Init</td>
<td>plapi_init, plapi_init_, PLAPI_INIT, PLAPI_INIT_</td>
</tr>
<tr>
<td>LAPI_Msg_string</td>
<td>PLAPI_Msg_string</td>
<td>plapi_msg_string, plapi_msg_string_, PLAPI_MSG_STRING, PLAPI_MSG_STRING_</td>
</tr>
</tbody>
</table>
### Table 21. LAPI profiling interfaces (continued)

<table>
<thead>
<tr>
<th>LAPI subroutine</th>
<th>C and C++ profiling interface</th>
<th>FORTRAN profiling interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPI_Msgpoll</td>
<td>PLAPI_Msgpoll</td>
<td>plapi_msgpoll, plapi_msgpoll_, PLAPI_MSGPOoll, PLAPI_MSGPOoll_</td>
</tr>
<tr>
<td>LAPI_Nopoll_wait</td>
<td>PLAPI_Nopoll_wait</td>
<td>plapi_nopoll_wait, plapi_nopoll_wait_, PLAPI_NOPOLL_WAIT, PLAPI_NOPOLL_WAIT_</td>
</tr>
<tr>
<td>LAPI_Probe</td>
<td>PLAPI_Probe</td>
<td>plapi_probe, plapi_probe_, PLAPI_PROBE, PLAPI_PROBE_</td>
</tr>
<tr>
<td>LAPI_Purge_totask</td>
<td>PLAPI_Purge_totask</td>
<td>plapi_purge_totask, plapi_purge_totask_, PLAPI_PURGE_TOTASK, PLAPI_PURGE_TOTASK_</td>
</tr>
<tr>
<td>LAPI_Put</td>
<td>PLAPI_Put</td>
<td>plapi_put, plapi_put_, PLAPI_PUT, PLAPI_PUT_</td>
</tr>
<tr>
<td>LAPI_Putv</td>
<td>PLAPI_Putv</td>
<td>plapi_putv, plapi_putv_, PLAPI_PUTV, PLAPI_PUTV_</td>
</tr>
<tr>
<td>LAPI_Qenv</td>
<td>PLAPI_Qenv</td>
<td>plapi_qenv, plapi_qenv_, PLAPI_QENV, PLAPI_QENV_</td>
</tr>
<tr>
<td>LAPI_Resume_totask</td>
<td>PLAPI_Resume_totask</td>
<td>plapi_resume_totask, plapi_resume_totask_, PLAPI_RESUME_TOTASK, PLAPI_RESUME_TOTASK_</td>
</tr>
<tr>
<td>LAPI_Rmw</td>
<td>PLAPI_Rmw</td>
<td>plapi_rmw, plapi_rmw_, PLAPI_RMW, PLAPI_RMW_</td>
</tr>
<tr>
<td>LAPI_Rmw64</td>
<td>PLAPI_Rmw64</td>
<td>plapi_rmw64, plapi_rmw64_, PLAPI_RMW64, PLAPI_RMW64_</td>
</tr>
<tr>
<td>LAPI_Senv</td>
<td>PLAPI_Senv</td>
<td>plapi_senv, plapi_senv_, PLAPI_SENV, PLAPI_SENV_</td>
</tr>
<tr>
<td>LAPI_Setcntr</td>
<td>PLAPI_Setcntr</td>
<td>plapi_setcntr, plapi_setcntr_, PLAPI_SETCNTR, PLAPI_SETCNTR_</td>
</tr>
<tr>
<td>LAPI_Setcntr_wstatus</td>
<td>PLAPI_Setcntr_wstatus</td>
<td>plapi_setcntr_wstatus, plapi_setcntr_wstatus_, PLAPI_SETCNTR_WSTATUS, PLAPI_SETCNTR_WSTATUS_</td>
</tr>
<tr>
<td>LAPI_Term</td>
<td>PLAPI_Term</td>
<td>plapi_term, plapi_term_, PLAPI_TERM, PLAPI_TERM_</td>
</tr>
<tr>
<td>LAPI_Util</td>
<td>PLAPI_Util</td>
<td>plapi_util, plapi_util_, PLAPIUTIL, PLAPIUTIL_</td>
</tr>
<tr>
<td>LAPI_Waitcntr</td>
<td>PLAPI_Waitcntr</td>
<td>plapi_waitcntr, plapi_waitcntr_, PLAPI_WAITCNTR, PLAPI_WAITCNTR_</td>
</tr>
<tr>
<td>LAPI_Xfer</td>
<td>PLAPI_Xfer</td>
<td>plapi_xfer, plapi_xfer_, PLAPI_XFER, PLAPI_XFER_</td>
</tr>
</tbody>
</table>

### Performing name-shift profiling

To use name-shift profiling routines that are either written to the C bindings with a LAPI program written in C, or that are written to the FORTRAN bindings with a LAPI program written in FORTRAN, use the following steps.

Programs that use LAPI’s C language bindings can create profiling libraries using the name-shifted interface.

- If you are both the creator and user of the profiling library and you are not using FORTRAN, follow steps 1 through 6. If you are using FORTRAN, follow steps 1 through 4, then steps 7 through 9.
- If you are the creator of the profiling library, follow steps 1 through 4. You also need to provide the user with the file created in step 3.
- If you are the user of the profiling library and you are not using FORTRAN, follow steps 5 and 6. If you are using FORTRAN, start at step 7. You will need to make sure that you have the file generated by the creator in step 3.

To perform LAPI name-shift profiling, follow the appropriate steps:
1. Create a source file that contains profiling versions of all of the LAPI subroutines you want to profile. For example, create a source file called `myprof_r.c` that contains the following code:

```c
#include <pthread.h>
#include <stdio.h>
#include <lapi.h>

int LAPI_Init(lapi_handle_t *hndl, lapi_info_t *lapi_info) {
    int rc;
    printf("Hello, from profiling layer LAPI_Init ...
");
    rc = PLAPI_Init(hndl, lapi_info);
    printf("goodbye, from profiling layer LAPI_Init ...
");
    return rc;
}

int lapi_init(lapi_handle_t *hndl, lapi_info_t *lapi_info, int *err) {
    int rc;
    printf("Hello, from profiling layer LAPI_Init ...
");
    rc = plapi_init(hndl, lapi_info, err);
    printf("goodbye, from profiling layer LAPI_Init ...
");
    return rc;
}
```

2. Compile the source file that contains your profiling LAPI routines. Use the `-I` flag to specify the location of `lapi.h`. For example, to compile the profiling source file you created in step 1:

```bash
c_c_r -c myprof_r.c -I/usr/include
```

3. Create an export file that contains all of the symbols your profiling library will export. Begin this file with the name of your profiling library and the name of the `.o` file that will contain the object code of your profiling routines. For example, to create an export file for the profiling source file that you created in step 1, create a file called `myprof_r.exp` that contains this statement:

```plaintext
LAPI_Init
```

4. Create a shared library called `libmyprof_r.a` that contains the profiled versions, exporting their symbols and linking with the LAPI library, using `myprof_r.exp` as shown. Use the `-l` flag to identify the LAPI profiling libraries and the `-L` flag to specify their locations. For example:

```bash
ld -o newmyprof_r.o myprof_r.o -bnoentry -bE:myprof_r.exp -lc_r -L/usr/lib\ -llapi_r -lpthreads
ar rv libmyprof_r.a newmyprof_r.o
```

5. Link your user program:

```bash
mpcc_r -o test1 test1.c -L. -lmyprof_r
```

6. Run the resulting executable.

7. For a FORTRAN program using a LAPI function that you have profiled in step 1, create a file called `hwinit.f` that contains the following statements:

```fortran
#include 'lapif.h'
integer :: handle
integer :: info(25)
integer :: ierror, i

do i=1, 25
    info(i) = 0
enddo
call LAPI_INIT(handle, info, ierror)
write(*,*) ierror
call LAPI_TERM(handle, ierror)
stop
end
```

8. Compile your FORTRAN program (`hwinit.f`) using the new library (`libmyprof_r.a`):

```bash
mpxlf_r -o hwinit hwinit.f -L. -lmyprof_r
```

9. Run the resulting executable.
A sample profiling program

A sample profiling program follows. Before it is linked with the LAPI library, the user program should be linked with a profiling library that is formed from the following program, using the instructions in "Performing name-shift profiling" on page 86. Then, the time spent in LAPI_Xfer and LAPI_Msgpoll will be printed out when the user program terminates. Note that this sample program is not multithread-safe.

```c
#include <stdio.h>
#include <sys/time.h>
#include <lapi.h>

typedef struct timeval prof_time_t;

prof_time_t xfer_time;
prof_time_t poll_time;

void accumulate(prof_time_t *sum, prof_time_t *start, prof_time_t *stop)
{
    sum->tv_sec += stop->tv_sec - start->tv_sec;
    sum->tv_usec += stop->tv_usec - start->tv_usec;
    if (sum->tv_usec >= 1000000) {
        sum->tv_usec -= 1000000;
        sum->tv_sec += 1;
    }
    if (sum->tv_usec < 0) {
        sum->tv_usec += 1000000;
        sum->tv_sec -= 1;
    }
}

int LAPI_Init(lapi_handle_t *hndl, lapi_info_t *lapi_info)
{
    bzero(&xfer_time, sizeof(xfer_time));
    bzero(&poll_time, sizeof(poll_time));
    return PLAPI_Init(hndl, lapi_info);
}

int LAPI_Xfer(lapi_handle_t hndl, lapi_xfer_t *xfer_cmd)
{
    int rc;
    prof_time_t start, stop;
    gettimeofday(&start, NULL);
    rc = PLAPI_Xfer(hndl, xfer_cmd);
    gettimeofday(&stop, NULL);
    accumulate(&xfer_time, &start, &stop);
    return rc;
}

int LAPI_Msgpoll(lapi_handle_t hndl, uint cnt, lapi_msg_info_t *info)
{
    int rc;
    prof_time_t start, stop;
    gettimeofday(&start, NULL);
    rc = PLAPI_Msgpoll(hndl, cnt, info);
    gettimeofday(&stop, NULL);
    accumulate(&poll_time, &start, &stop);
    return rc;
}

int LAPI_Term(lapi_handle_t hndl)
{
    printf("LAPI_Xfer time: %u h %u m %u s %06u us\n",
            xfer_time.tv_sec/3600, xfer_time.tv_sec%3600/60,
            xfer_time.tv_usec/1000000,
            xfer_time.tv_usec%1000000/1000,
            xfer_time.tv_usec%1000000%1000/100,
            xfer_time.tv_usec%1000000%1000%100/100);
}
```
xfer_time.tv_sec%60, xfer_time.tv_usec);
printf("LAPI_Msgpoll time: %uh %um %us %06us
",
poll_time.tv_sec/3600, poll_time.tv_sec%3600/60,
poll_time.tv_sec%60, poll_time.tv_usec);
return PLAPI_Term(hndl);
Chapter 11. Compiling LAPI programs

This chapter shows you how to compile LAPI programs on systems that are running PE. See Chapter 17, “Using LAPI on a standalone system,” on page 141 for information about compiling LAPI programs on systems that are not running PE.

On an AIX system

As with a serial application, you must compile a parallel C, C++, or FORTRAN program before you can run it. The commands shown in Table 22 and Table 23 link in the POE partition manager and support programs that use the threaded LAPI library. The commands shown in Table 22 also link in the AIX communication subsystem (CSS) interfaces.

Table 22 shows what commands to enter to compile an RSCT LAPI for AIX program on an AIX system that is running PE.

Table 22. Compiling LAPI programs on a system running PE for AIX

<table>
<thead>
<tr>
<th>To compile a C program:</th>
<th>mpcc_r program.c -o program</th>
</tr>
</thead>
<tbody>
<tr>
<td>To compile a C++ program:</td>
<td>mpCC_r program.C -o program</td>
</tr>
<tr>
<td>To compile a FORTRAN program:</td>
<td>mpfl_r program.f -o program</td>
</tr>
</tbody>
</table>

If you compiled your program using one of the commands shown in Table 22, the CSS libraries are dynamically linked with the executable when you run your program. Subroutines in these libraries enable POE’s home node (the node from which the parallel program is invoked) to communicate with the parallel tasks, and tasks with each other.

On a Linux system

Table 23 shows what commands to enter to compile a LAPI program on a Linux system that is running PE.

Table 23. Compiling LAPI programs on a system running PE for Linux

<table>
<thead>
<tr>
<th>To compile a C program:</th>
<th>mpcc program.c -o program</th>
</tr>
</thead>
<tbody>
<tr>
<td>To compile a C++ program:</td>
<td>mpCC program.C -o program</td>
</tr>
<tr>
<td>To compile a FORTRAN program:</td>
<td>mfort program.f -o program</td>
</tr>
</tbody>
</table>
### Chapter 16. Threaded programming

- General guidelines ............................................................ 137
- Using LAPI_Address_init ...................................................... 137
- Making global fence calls .................................................... 138
- Making "wait on counter" calls .............................................. 138
- Synchronizing threads across tasks ..................................... 138
- Using handlers .................................................................... 138
- LAPI threads ....................................................................... 139

### Chapter 17. Using LAPI on a standalone system

- Setting up a standalone system ............................................. 141
- Initializing a standalone system .......................................... 143
  - In UDP/IP mode ................................................................. 143
  - In US mode on AIX ............................................................ 143
- Compiling LAPI programs on a standalone system ............ 144
- Standalone systems: restrictions ......................................... 144
Chapter 12. Advanced programming

This book includes several topics related to advanced programming with LAPI, such as:

- "Extended user header support" on page 23
- "The enhanced header handler interface"
- "Inline handlers" on page 97
- "Optimizing communication for small messages" on page 98
- "32-bit and 64-bit interoperability" on page 101

The enhanced header handler interface

In PSSP LAPI, the header handler was originally implemented as follows:

```c
typedef void *(hdr_hndlr_t)(lapi_handle_t *hndl, void *uhdr, uint *uhdr_len,
    ulong *msg_len, compl_hndlr_t **compl_h, void **uinfo);
```

LAPI passes the LAPI context, user header address, user header length, and message length into a header handler. From the header handler, you can pass back to LAPI a data buffer address, a completion handler, and any user-defined information for the completion handler.

In order to allow more information exchange between LAPI and the user program, the `msg_len` parameter type has been changed from `ulong *` to `lapi_return_info_t *`, which is a pointer to the following structure:

```c
typedef struct {
    ulong msg_len;
    int MAGIC;
    lapi_ret_flags_t ret_flags;
    lapi_ctl_flags_t ctl_flags;
    lapi_dg_handle_t dgsp_handle;
    ulong bytes;
    int src;
    void * udata_one_pkt_ptr;
    ulong recv_offset_dgsp_bytes;
} lapi_return_info_t;
```

**Note:** This applies to new or modified LAPI programs only.

With this enhanced interface, LAPI has the following new capabilities:

- running the completion handler inline as opposed to in a separate thread, which significantly improves performance (see "Inline handlers" on page 97)
- options for the user to drop the message instead of delivering it (described in this section)
- transferring data in the layout described by data gather-scatter programs (DGSPs) (see "Using data gather/scatter programs (DGSPs)" on page 52)
- optimization for receiving one-packet messages (see "Receive-side optimization for single-packet messages" on page 106)

This extension does not require any change in existing LAPI programs that were coded and compiled with the original header handler interface. To these programs, the header handler interface remains the same because 1) `msg_len` is the first field in the `lapi_return_info_t` structure and 2) all other fields are set to default values by LAPI before calling a header handler, so LAPI behaves exactly the same as it does without the extension if none of the fields are changed by the header handler.
The prototype of the header handler remains the same in LAPI's header files (as it is shown above), but you can optionally use the following extended interface to define header handlers:

```c
void *header_handler(lapi_handle_t *hndl, void *uhdr, uint *uhdr_len,
    lapi_return_info_t *msg_len, compl_hndlr_t **compl_h, void **uinfo);
```

LAPI allocates the `lapi_return_info_t` structure, initializes it with appropriate values, and passes a pointer to the structure into the header handler. You can then alter the fields of this structure as needed to convey specific information back to LAPI.

The `lapi_return_info_t` structure includes the following fields:

<table>
<thead>
<tr>
<th>Field (input or output)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>msg_len</code> (IN)</td>
<td>Specifies the length of the incoming message (for compatibility with the original header handler interface).</td>
</tr>
<tr>
<td><code>MAGIC</code> (IN)</td>
<td>Indicates the integrity of the structure. The user must not change this field.</td>
</tr>
<tr>
<td><code>ret_flags</code> (OUT)</td>
<td>Tells LAPI how to run the completion handler (if any). See [&quot;Setting the ret_flags field&quot;](page 97) for a list of valid values.</td>
</tr>
<tr>
<td><code>ctl_flags</code> (OUT)</td>
<td>Tells LAPI to deliver, &quot;bury&quot;, or drop the incoming message. See [&quot;Setting the ctl_flags field&quot;](page 97) for a list of valid values.</td>
</tr>
<tr>
<td><code>dgsp_handle</code> (OUT)</td>
<td>Tells LAPI to deliver the message according to the specified DGSP. If this field is NULL (the default), LAPI delivers the message to the contiguous buffer returned by the header handler.</td>
</tr>
<tr>
<td><code>bytes</code> (OUT)</td>
<td>Tells LAPI the number of bytes to deliver when a DGSP is used. This value can be less than the message length, but should not be greater.</td>
</tr>
<tr>
<td><code>src</code> (IN)</td>
<td>Specifies the source task ID of the message.</td>
</tr>
<tr>
<td><code>udata_one_pkt_ptr</code> (IN)</td>
<td>Specifies the pointer to the incoming data if this is a one-packet message. Otherwise, the value is set to NULL.</td>
</tr>
<tr>
<td><code>recv_offset dgsp bytes</code> (OUT)</td>
<td>Specifies the initial offset bytes that the target DGSP runs to get the DGSM in the correct state before processing the user's data. Otherwise, the value is set to 0. This field only applies when LAPI_Xfer with transfer type LAPI_AMX_XFER (AMX) is used.</td>
</tr>
</tbody>
</table>

### Setting the ret_flags field

You can set `ret_flags` to the following values to instruct LAPI how to call the completion handler for the incoming message:

- **LAPI_NORMAL**
  - Instructs LAPI to run the completion handler in the completion handler thread.
- **LAPI_LOCAL_STATE**
  - Instructs LAPI to run the completion handler in the...
same thread that is receiving the message. (You
guarantee that the completion handler does not
make any LAPI calls.)

**LAPI_SEND_REPLY**

Instructs LAPI to run the completion handler in the
same thread that is receiving the message. (You
want to send a reply to the message by calling
LAPI.)

### Setting the ctl Flags field

You can set the `ctl_flags` field to instruct LAPI how to handle the incoming message besides delivering it:

**LAPI_BURY_MSG**

does not copy data for the message. (From the
perspective of the origin side, there is no difference
between this setting and **LAPI_DELIVER_MSG**.)

**LAPI_DELIVER_MSG**

delivers the message as normal (the default).

**Note:** You *cannot* make LAPI calls from within the header handler. For contiguous
data, you can copy the data to the buffer you specified. For non-contiguous data,
you must pass the DGSP handle and a buffer address to LAPI. LAPI will unpack
the data to the specified buffer address.

### Inline handlers

You can prioritize the completion of certain messages by requesting that your
header handler, send completion handler, or receive completion handler be run
inline. By default, you can call LAPI functions inline within the header handler and
the send completion handler. The receive completion handler can call LAPI
functions as well.

LAPI runs the receive completion handler inline (if possible) if you set the `ret_flags`
field in the `lapi_return_info_t` structure to **LAPI_SEND_REPLY** or
**LAPI_LOCAL_STATE**. This flag is returned to LAPI as a reference parameter in the
header handler you provided.

If you request that your receive completion handler be run inline, LAPI tries to
acquire the necessary send token before execution of the receive completion
handler is attempted. If you specified **LAPI_SEND_REPLY**, a check is made for
send tokens. Only in the case of **LAPI_SEND_REPLY** is it necessary to check for
send tokens, because you must not call a LAPI routine in the receive completion
handler in the case of **LAPI_LOCAL_STATE**.

If LAPI is successful in getting its internal send token without polling for a token to
free up, or if you specified **LAPI_LOCAL_STATE**, the receive completion handler is
run inline. Otherwise, LAPI enqueues the receive completion handler in the
separate receive completion handler thread. LAPI does not allow any normal LAPI
functions to be called within it (that is, by the completion handler thread). See
[Figure 11 on page 98](#).
See [LAPI_Amsend](#) on page 168 for more information and an example.

I/O operations and blocking calls, including blocking LAPI calls, should **not** be performed within a header handler, send completion handler, or receive completion handler that is run inline. Inline handlers should be short, because no progress can be made while the main thread is running the handler. You must use caution with inline handlers so that LAPI's internal queues do not fill up while waiting for the handler to complete.

### Optimizing communication for small messages

In order to optimize communication for very small messages, LAPI supports a new generic transfer function: **LAPI_Xfer** with transfer type **LAPI_AM_LW_XFER** (**AM_LW**). This function, which provides the the lowest latency (and thus, the fastest communication), can be used for messages as large as 128 bytes. This size includes the user data and any user header. In this function, several restrictions are placed on usage — counters are not supported, for example. These restrictions allow for simplifications in the protocol path, thus enabling better performance.

**AM_LW** uses the same interface as **AM**, but ignores these fields: **shdlr, sinfo, tgt_cntr, org_cntr, and cmpl_cntr**. See [LAPI_AM_LW_XFER](#) on page 264 for more information.

Restrictions for **AM_LW** follow.

On the sending side:
- The sum of **uhdr_len** and **udata_len** in **lapi_am_t** must not exceed 128 bytes.
- The function does not support the use of any counters. Any specified counters are ignored.
- There is no send completion handler support and LAPI guarantees that buffers for user header and user data can be reused right after the function returns.
- **hdr_hdl** must be an index to the function table, as defined by **LAPI_Addr_set**.

On the receiving side, the LAPI receive function only updates the **msg_len**, **src**, and **udata_one_pkt_ptr** fields in the **lapi_return_info_t** structure before calling the user's header handler. The user must consume the data pointed by...
udata_one_pkt_ptr in the header handler and LAPI will not copy the data for the user. The return value of the header handler is always ignored. Unlike AM, using AM_LW guarantees that the receive completion handler, if provided by the user, is always executed inline. (For AM, you must set ret_flags to LAPI_LOCAL_STATE or LAPI_SEND_REPLY for the completion handler to be executed inline.) The following lapi_return_info_t fields are not supported: ret_flags, ctl_flags, and dgsp_handle. For more information about lapi_return_info_t, see “The enhanced header handler interface” on page 95.

A small sample program that shows how to use AM_LW follows, as well the related considerations and restrictions. In the program, task PIDs are exchanged between even-numbered and odd-numbered tasks by calling LAPI_Xfer with the LAPI_AM_LW_XFER transfer type.

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <lapi.h>

#define RC(statement) 
{ 
  int rc = statement; 
  if (rc != 0) { 
    printf(#statement " rc = %d, line %d
", rc, __LINE__); 
    exit(-1); 
  } 
}

lapi_handle_t hndl;
int myid;
int complete = 0;

/* Completion handler */
void
am_complete(lapi_handle_t *hndl, void *saved_info)
{
  pid_t *partner_pid = (pid_t *)saved_info;

  /* This completion handler for AM_LW is always executed inline, 
  so the rules of inline completion must be followed. */
  printf("Task %d: Completion handler executed in thread %d\n", 
    myid, pthread_self());
  printf("Task %d: Partner's PID is %d\n", 
    myid, *(pid_t *)saved_info);
  /* Free allocated buffer and mark transfer complete. */
  free(partner_pid);
  complete = 1;
}

/* Header handler */
void *
am_hndlr(lapi_handle_t *hndl, int *u_hdr, uint *hdr_len,
  lapi_return_info_t *ret_info,
  compl_hndlr_t **chndlr, void **saved_info)
{
  pid_t *partner_pid;

  /* User data must be copied/consumed here. LAPI_AM_LW_XFER will not 
  copy data to a user-provided buffer. */
  partner_pid = malloc(sizeof(pid_t));
  *partner_pid = *(pid_t *)(ret_info->udata_one_pkt_ptr);

  /* Completion handler of AM_LW is always executed inline, so there is 
  no need to set a flag in ret_info to indicate this. */
*chndlr = am_complete;
*saved_info = partner_pid;
printf("Task %d: Header handler executed in thread %d\n",
myid, pthread_self());

/* Return value is ignored by LAPI because data has been copied/consumed. */
return NULL;
}

main(int argc, char *argv[])
{
    lapi_info_t lapi_info;
    pid_t pid;
    lapi_xfer_t xfer;
    lapi_msg_info_t msg_info;

    /* Initialize LAPI */
bzero(&lapi_info, sizeof(lapi_info));
    RC( LAPI_Init(&hndl, &lapi_info) );
    RC( LAPI_Qenv(hndl, TASK_ID, &myid) );

    /* Set address index for header handler. */
    RC( LAPI_Addr_set(hndl, (void*)&am_hndlr, 1) );

    /* Send my PID to my partner */
    pid = getpid();
    printf("Task %d: my PID is %d\n", myid, pid);
    bzero(&xfer, sizeof(xfer));
    xfer.Xfer_type = LAPI_AM_LW_XFER;
    xfer.Am.flags = 0;
    xfer.Am.hdr_hdl = 1; /* AM_LW supports only address index */
    xfer.Am.tgt = myid ^ 1;
    xfer.Am.uhdr = NULL;
    xfer.Am.uhdr_len = 0;
    xfer.Am.udata = &pid;
    xfer.Am.udata_len = sizeof(pid);
    /* No need to set other fields because LAPI_AM_LW_XFER ignores them. */

    /* LAPI_AM_LW_XFER supports message size no larger than 128 bytes. */
    assert(xfer.Am.uhdr_len + xfer.Am.udata_len <= 128);

    /* Start LAPI_AM_LW_XFER transfer. LAPI guarantees that buffers for user
    header and user data are reusable upon the return of LAPI_Xfer. */
    RC( LAPI_Xfer(hndl, &xfer) );

    /* Poll until the information from partner is received. */
    while (!complete)
    RC( LAPI_Msgpoll(hndl, 1000, &msg_info) );

    /* Terminate LAPI */
    RC( LAPI_Term(hndl) );
}

The output looks like this:
Task 0: my PID is 70570
Task 1: my PID is 38872
Task 0: Header handler executed in thread 1
Task 0: Completion handler executed in thread 1
Task 0: Partner's PID is 38872
Task 1: Header handler executed in thread 1
Task 1: Completion handler executed in thread 1
Task 1: Partner's PID is 70570
32-bit and 64-bit interoperability

The `lapi_long_t` datatype
The `lapi_long_t` datatype is used by various LAPI functions to ensure 32-bit/64-bit interoperability.

The `LAPI_Address_init64` subroutine
The `LAPI_Address_init64` subroutine provides a mechanism for a 64-bit task to share addresses with 32-bit tasks. See "LAPI_Address_init64" on page 166 for more information.

The `LAPI_Xfer` interface
The `LAPI_Xfer` interface serves as a wrapper function for LAPI data transfer functions. Its remote address fields are expanded to be of type `lapi_long_t`, which is long enough for a 64-bit address. This allows a 32-bit task to send data to 64-bit addresses, which may be important in client/server programs. See "LAPI_Xfer" on page 260 for more information.
Chapter 13. LAPI performance considerations

LAPI provides a one-sided programming model with a pseudo shared-memory view of multi-task operations on distributed servers. For this reason, it is expected that users will find the best performance when LAPI is used with interrupts turned on (the default value). Indeed, LAPI message-passing calls communicate with a remote task without the need for that task’s active participation in the communication. The use of interrupts frees both the origin and target-side tasks from the need to poll for buffer availability and message completion, respectively.

Use of handlers

There are a few points in LAPI’s message transfer sequence in which user notification can occur. In most instances, LAPI provides notification through the incrementing of a counter or the invocation of a user-supplied handler. LAPI’s one-sided model tends to perform better through the use of handlers for notification, because they involve the actual execution of user code without any explicit action on the part of the task in question. The use of counters requires the task to poll on the counter value. This could needlessly consume CPU cycles, depending on what else the application has available for the CPU to work on and how counter checking is performed.

One counter-example to the philosophy that the use of handlers provides better performance is as follows. When running in a fully-loaded configuration (n tasks on n CPUs), LAPI may perform better in polling mode (interrupts off). One explanation for this anomaly has to do with the scheduling of the thread that handles interrupts.

Running in interrupt mode

LAPI runs in interrupt mode by default. Interrupts cause thread context switching, which can adversely affect performance. For optimal performance, the user program should turn off interrupt mode when entering a polling section and turn it on again when exiting the polling section. To turn interrupt mode off and on, call LAPI_Senv with the INTERRUPT_SET attribute. See “LAPI_Senv” on page 234 for more information. Whenever the user program transfers control to LAPI by calling a LAPI function, LAPI will drive communication traffic, so interrupts are not necessary. Interrupts will only introduce the overhead of thread context switching if LAPI is already polling.

Enabling receive interrupt and using CPU affinity, that is, binding one task to one CPU, could impact performance. The contention between the main thread and the interrupt thread on a single CPU creates delays that could have an impact on LAPI_Address_init performance. The impact could be greater if MP_EUIDEVICE=sn_all because there is one receive interrupt thread per striping instance. If you want to run with CPU affinity, disable receive interrupt using LAPI_Senv with the INTERRUPT_SET attribute. If you need receive interrupt to be enabled, set MP_TASK_AFFINITY=MCM (MCM affinity) or MP_TASK_AFFINITY=-1 (no affinity). For more information about managing task affinity, see PE: Operation and Use. Also, run the vmstat 1 command and check the number of runnable threads in the first column of the command output. If this number is larger than the number of CPUs, the system will be overloaded, which will impact performance.
For RSCT LAPI for AIX jobs running in interrupt mode, performance gains may be realized by setting environment variables `AIXTHREAD_SCOPE=S` and `RT_GRQ=ON`. There is also evidence to suggest that the performance of \( n-1 \) tasks on \( n \) CPUs is considerably better than that of a fully-loaded system.

LAPI for Linux does not perform well when running in interrupt mode.

---

### Running in UDP/IP mode

If you are running RSCT LAPI for AIX or LAPI for Linux:

- **Over IP** —
  1. Turn off interrupt coalescing for lower latency. See the tuning documentation for your particular operating system for information about how to do this.
  2. To enable jumbo frame Ethernet for higher bandwidth, enter:

     ```
     ifconfig interface_name mtu 9000 up
     ```

     This enables the use of 9000-byte maximum transmission unit (MTU) Ethernet instead of the default 1500-byte MTU Ethernet.

- **On a higher-bandwidth network with a minimal number of packet drops**, consider using the `MP_UDP_PACKET_SIZE` environment variable. This tuning option can provide applications with higher bandwidth capability. The higher the capacity of the network, the greater the potential for higher bandwidth as the value of `MP_UDP_PACKET_SIZE` increases. There is also the potential for packet drops associated with reassembling IP fragments when `MP_UDP_PACKET_SIZE` is set higher than the MTU of the network.

On Linux, there can be great benefit to making system tunable changes that allow for more buffers to be allocated for IP reassembly. To get an idea if packets might be dropped due to insufficient IP reassembly buffers, review the following counters from `netstat -s`:

```
# netstat -s
Ip:
  0 fragments dropped after timeout
  [. . .]
  0 packet reassemblies failed
```

If setting `MP_UDP_PACKET_SIZE` higher than the network MTU results in an increase of the “packet reassembles failed” counter as a LAPI job runs, the following `/proc` filesystem tunables can be increased to see if performance improves:

```
/proc/sys/net/ipv4/ipfrag_low_thresh (default = 196608)
/proc/sys/net/ipv4/ipfrag_high_thresh (default = 262144)
```

For example, on Gigabit Ethernet, the following commands might improve performance with higher `MP_UDP_PACKET_SIZE` values (16384, for example):

```
echo 524288 > /proc/sys/net/ipv4/ipfrag_low_thresh
echo 589824 > /proc/sys/net/ipv4/ipfrag_high_thresh
```

Note that even higher “ipfrag thresh” values may be applicable to higher-bandwidth networks, such as InfiniBand. If, after experimenting with the `ipfrag_low_thresh` and `ipfrag_high_thresh` tunables, the `netstat` “packet reassembles failed” counter still increments and the number of fragments dropped after timeout continues to increase, there is another Linux `/proc` filesystem tunable that can be changed:

```
/proc/sys/net/ipv4/ipfrag_time (default = 30)
```
The `ipfrag_time` tunable can be increased to see if performance can be
improved. For example:
\[
\text{echo 60 > /proc/sys/net/ipv4/ipfrag_time}
\]

On AIX, a similar timeout and counter exists for fragments that arrive too far
apart. The AIX `netstat -s` command shows a "fragments dropped after timeout"
counter. If this increases during the running of a LAPI job when `MP_UDP_PACKET_SIZE`
is set higher than the network MTU, the timeout
allowed for fragment reassembly can be increased with the `no` command. For
example:
\[
\text{no -o ipfragttl = 4 (default = 2)}
\]

**Send space and receive space**

When LAPI is running over IP, it requires a certain amount of "send space" to buffer
packets to transmit and a certain amount of "receive space" to buffer packets to
receive. When there is not enough receive space to hold incoming packets, packets
will be dropped, causing performance degradation.

On AIX, you can change the amount of send space by modifying the `udp_sendspace`
attribute of the `no` command. You can change the amount of receive space by modifying the `udp_recvspace`
attribute of the `no` command. Send space and receive space are upper-bounded by the `sb_max`
attribute of the `no` command. The key tuning factor is to have a sufficiently large `sb_max`
value. This value depends on the total number of tasks and the communication pattern in the
job. In general, the value of `sb_max` should be equal to the maximum number of
tasks that simultaneously communicate with one task multiplied by the value of `MP_UDP_PACKET_SIZE`
(see "Tunable environment variables" on page 107 for
more information).

On Linux, log in as `root` and make the following changes:

1. To increase the send space of a socket to 1MB, enter:
   \[
   \text{echo 1048576 > /proc/sys/net/core/wmem_max}
   \]
2. To increase the receive space of a socket to 8MB, enter:
   \[
   \text{echo 8388608 > /proc/sys/net/core/rmem_max}
   \]

**Running with shared memory enabled**

If you are using LAPI shared memory on Linux (running over IP or US), you should
increase the maximum shared memory size to 256MB. To do this, enter:
\[
\text{echo 268435456 > /proc/sys/kernel/shmmax}
\]

**User header data**

When you use the user header facility for LAPI message-passing calls, make sure
the user header length is a multiple of 4 (that is, an integral number of words). Due
to the layout of data within the lower-level first-in, first-out structures (FIFOs), you
will see noticeable performance gains if header lengths are an integral number of
words.
Send-side copy of small messages

Because LAPI is a reliable protocol, it must handle retransmission if packets are dropped by the lower communication layers. To do this, LAPI must maintain user data uncorrupted until it receives acknowledgement that all packets have arrived. Because LAPI calls are non-blocking, return from a LAPI call does not mean that the send-side data buffer can be modified by the sending task.

LAPI provides the origin counter and send completion handler for this notification. For small messages however, LAPI will make a local copy of the message so that the origin buffer is reusable immediately upon return by the sending task. When sending such messages, the user can assume that the send-side data buffer is available for modification immediately upon return of the LAPI call. The maximum size for this local copy can be set using the `MP_REXMIT_BUF_SIZE` environment variable. The default is to perform the copy for messages of size 128 bytes or less, counting both message data and any user header.

Receive-side optimization for single-packet messages

For single-packet messages, you can optimize the copying of data out of the network FIFO on the receive side. LAPI provides a mechanism with which you can copy the data directly from the receive FIFO. For single packets, LAPI passes the data address to your header handler using the `msg_len` parameter of the `lapi_return_info_t` mechanism. You can then access the data directly in the receive FIFO. If you choose to copy data directly from the FIFO, it can inform LAPI that the message does not need to be delivered by returning NULL from the header handler without changing `lapi_return_info_t`. Recall that the header handler returns the base address of the target data buffer. A NULL value indicates to LAPI that no data is to be copied. An alternative way to indicate to LAPI to not copy the message data is to set the `ctl_flag` field in `lapi_return_info_t` to `LAPI_BURY_MSG`. You must contain the data from the receive FIFO within the header handler when using this optimization. Upon header handler return, LAPI informs the communication subsystem that the receive FIFO slot is available for modification.

Here is a sample header handler that uses this method for fast retrieval of a single-packet message:

```c
void *header_handler(lapi_handle_t *hdl, void *uhdr, uint *uhdr_len,
                     ulong *ret_info, compl_hndlr_t **compl_h, void **uinfo)
{
    lapi_return_info_t *ret_info_ptr; /* struct pointer */
    void *base_addr; /* base address to pull from, could also be */
             /* a pointer to a user-defined datatype */
    ulong msg_len; /* first field of struct is the msg length */

    /* grab struct */
    ret_info_ptr = (lapi_return_info_t *) ret_info;
    base_addr = (void *)(ret_info_ptr->udata_one_pkt_ptr);
    msg_len = ret_info_ptr->msg_len;

    /* process data at base_addr */
    /* inform LAPI not to copy data */
    return NULL;
}
```
On-node barrier synchronization

For improved performance of on-node barrier synchronization, LAPI includes support for the barrier synchronization register (BSR). The BSR is a memory register that is located on IBM Power (POWER6) servers. It performs barrier synchronization, which is a method of synchronizing the threads in a parallel application. To use the BSR support, you must be running 64-bit programs on AIX 6.1. The `rsct.lapi.bsr` fileset must be installed, and you must enable the BSR support in order to use it. For information about installing and enabling this fileset, see Chapter 4, “Installing RSCT LAPI for AIX,” on page 29.

Any user attempting a read-modify-write operation on LAPI library-allocated storage could inadvertently affect the memory that is mapped to the BSR. Any such access will lead to unpredictable results.

Running multiple LAPI contexts per task

Each LAPI context (or LAPI instance) consumes a certain amount of system resources. Using multiple LAPI contexts per task (or process) multiplies the system resource usage by the number of LAPI contexts. When you increase the number of LAPI contexts per task by a certain factor, you need to correspondingly reduce the number of tasks per node by the same factor so that you don’t exceed the system resource limits on a node.

For example, suppose you had enough system resources to support running two LAPI contexts per task (`MP_MSG_API=mpi,lapi`) with previous releases of LAPI. With the current release of LAPI, if you want to run four LAPI contexts per task, you need to reduce the number of tasks per node by half.

Tunable environment variables

The following environment variables are considered user-tunable for performance:

**MP_ACK_THRESH**

Sets the number of packets that are received before LAPI returns a batch of acknowledgments to the sending task. If you do not set this variable, LAPI sets the default value according to the type of communication adapter that is being used.

*Note: If you decide to set this variable, proceed with caution.*

The value must be in the range \( 1 \leq MP\_ACK\_THRESH \leq 31 \). A value that is too small will result in high-acknowledgment traffic generated by the job consisting of a large number of small packets. A value that is too large may impede progress on the sending side and slow down the entire job, because send-side flow control will prevent the transmission of additional packets until packets are acknowledged.

**MP_BULK_MIN_MSG_SIZE**

(AIX only) Changes the minimum message size for which LAPI will attempt to make bulk transfers. This environment variable is a hint that may or may not be honored by the communication library.
MP_FIFO_MTU

Controls the maximum transmission unit (MTU) size for FIFO mode. The default is 2048. The other valid value is 4096. If the value is set to 4096, system administrators need to make sure that 4K MTU is enabled consistently on all of the InfiniBand switches in a cluster. Users need to get input from the system administrator about 4K MTU availability before enabling 4K MTU for their programs. Running with the 4K MTU option in protocols where it is not enabled in the switch can result in programs hanging. This environment variable is valid on Power 575 systems with the IBM GX Dual-Port 4x IB Host Channel Adapter.

MP_POLLING_INTERVAL

Controls the interval for LAPI timer pops (in microseconds). Timer pops cause LAPI to go through its acknowledgment and retransmit processing logic. The default is 400000 (400 milliseconds).

MP_RDMA_MTU

Controls the MTU size for RDMA mode. The default is 2048. The other valid value is 4096. If the value is set to 4096, system administrators need to make sure that 4K MTU is enabled consistently on all of the InfiniBand switches in a cluster. Users need to get input from the system administrator about 4K MTU availability before enabling 4K MTU for their programs. Running with the 4K MTU option in protocols where it is not enabled in the switch can result in programs hanging. This environment variable is valid on Power 575 systems with the IBM GX Dual-Port 4x IB Host Channel Adapter.

MP_RETRANSMIT_INTERVAL

Controls how often the communication subsystem library checks to see if it should retransmit packets that have not been acknowledged. The value specified is the number of polling loops between checks. The default is 1000000.

MP_REXMIT_BUF_CNT

Specifies the number of buffers that LAPI must allocate. The size of each buffer is defined by MP_REXMIT_BUF_SIZE. This count indicates the number of in-flight messages smaller than MP_REXMIT_BUF_SIZE that LAPI can store in its local buffers in order to free up the user’s message buffers more quickly. The default is 128.

MP_REXMIT_BUF_SIZE

Specifies the maximum message size, in bytes, that LAPI will store in its local buffers in order to more quickly free up the user buffer containing message data. This size indicates the size of the local buffers LAPI will allocate to store such messages, and will impact memory usage, while potentially improving...
performance. LAPI will use the buffer to store the user header and the user data. The default is 16384.

**MP_RFIFO_SIZE**

Controls the size, in bytes, of a receiving FIFO. For a job that has fewer than 2048 tasks, the default value is 4194304. For a job that has 2048 tasks or more, the default value is 16777216. If this environment variable is set, it overrides the default value.

**MP_UDP_PACKET_SIZE**

Controls the size of LAPI packets for UDP data transfer.

For optimum performance, this variable must be set to the size of the maximum transfer unit (MTU) of IP that is in use on the system. Setting **MP_UDP_PACKET_SIZE** to a value that is larger than the IP MTU size will result in potential performance degradation due to packetization by the IP layer for messages larger than **MP_UDP_PACKET_SIZE**. Setting it to a value that is smaller than the IP MTU size will result in unnecessary packetization overhead in the LAPI layer.

See “Variables for performance tuning” on page 320 for more information.
Chapter 14. Lock sharing

Sharing locks with LAPI provides increased efficiency in protocol layering and user programming. When you need to use a locking mechanism to protect your programs’ data structures, you can use the same locking mechanism that is employed by LAPI through its lock sharing interface. This way, your program is more tightly coupled with LAPI in terms of locking. When compared to the use of a separate lock, the use of a shared lock in your program may result in improved latency and throughput.

This optional LAPI function is used mainly to improve performance, especially latency. You can share one lock between LAPI and your program. If you decide not to use a shared lock, your program will proceed as if this function does not exist. If you want to use a shared lock, you need to understand what LAPI does and what you need to do.

LAPI uses locks to protect its internal data structures. Typically, LAPI API calls involve acquisition and release of the LAPI lock. Multi-threaded user programs also often use locks to protect data structures of the user program. It is recommended that such locks are released before making a LAPI API call, because any (unrelated) LAPI message may be received or completed before the API call completes. Such a reception or completion can result in the invocation of the user-provided LAPI handler functions that can access user structures and therefore, will need the user lock to be free for acquisition from within the user-provided handlers.

Having separate locks to protect user structures and LAPI’s internal data structures can result in significant inefficiencies due to locks needing to be released before LAPI calls, LAPI locks being acquired and released within the API call, and locks potentially being acquired and released within the body of user handler functions that are executed as a result of the API invocation. Such overhead can be significantly reduced by calling the LAPI_GET_THREAD_FUNC utility, which returns function pointers to various locking and signaling functions associated with LAPI’s internal lock, and thereafter use LAPI’s internal lock with these functions to protect data structures in your program. Thus, your program can share a lock with LAPI, using the lock to protect your data structures, while LAPI uses the same lock to protect its internal data structures. When you invoke LAPI API calls while holding the shared lock, LAPI will check to determine whether the lock is already held by the caller, and if so, continue with LAPI functions without acquiring the lock again. Similarly, when user handler functions are called, if they use the locking functions provided by LAPI_GET_THREAD_FUNC, LAPI can ensure that the lock is already held so the user doesn’t need to get the lock again. Such lock sharing can result in significant performance gains.

For more information, see LAPI_GET_THREAD_FUNC on page 247.

Scenarios without lock sharing

To understand lock sharing better, let’s consider the case without lock sharing, in which the user program uses its own lock to protect data structures. Figure 12 on page 112 and Figure 13 on page 113 illustrate the typical calling sequences.
Figure 12. A program initiates a call to LAPI, without lock sharing
In Figure 12 on page 112, the user program initiates a call to LAPI during some processing. The user program releases its lock before calling LAPI and reacquires the lock after LAPI returns. In Figure 13, LAPI initiates a callback (a header handler callback, for example), into the user program, without releasing LAPI's lock. To embed the acquisition of one lock inside another will not cause any deadlock problems as long as the locking hierarchy is strictly maintained. In these examples, a hierarchy is forced, so the LAPI lock will never be acquired inside the user lock. The user lock must always be released before LAPI is called.

This book refers to a user-initiated call sequence as a “down-call” and a LAPI-initiated call sequence as an “up-call”. Up-calls and down-calls can be embedded inside each other, as illustrated by Figure 14 on page 114 and Figure 15 on page 115.
Figure 14. A program initiates a call to LAPI, with embedded up- and down- calls
Without lock sharing, there are many lock acquisition and release pairs involved: three pairs in Figure 12 on page 112, two pairs in Figure 13 on page 113, four pairs in Figure 14 on page 114, and three pairs in Figure 15. When the cost of locking turns out to be critical for latency-sensitive applications, it would be much more desirable to reduce locking operations using the lock sharing function provided by LAPI.

**Scenarios with lock sharing**

Lock sharing makes the user program more tightly coupled with LAPI because one shared lock is now protecting both the user data structures and LAPI data structures. When the user program calls LAPI, it is optional to release the shared lock and to reacquire the lock after LAPI returns, as depicted in the down-call in Figure 16 on page 116. However, for optimal performance, it is better not to release the shared lock if the user has already acquired it. Because LAPI does not make any assumptions about the lock ownership, it will check the ownership upon entry into any of its API calls. If a lock is not held, LAPI will acquire the lock and release it after processing. Checking for lock ownership is much less expensive than getting a lock, so the locking cost is essentially one pair of lock acquisition and release, as shown in Figure 16 on page 116. For an up-call, LAPI guarantees that the shared lock is held before the user callback is invoked so there is no need to check lock ownership in the user callback. Similarly, the locking cost is
is also only one lock acquisition and release pair.

Figure 16. A program initiates a call to LAPI, with one lock acquisition and release pair
When up-calls and down-calls are embedded inside each other, the saving from lock sharing becomes more prominent. As shown in Figure 18 on page 118 and Figure 19 on page 119, only one lock acquisition and release pair is involved in both cases, compared to four pairs and three pairs without lock sharing, as shown in Figure 14 on page 114 and Figure 15 on page 115, respectively.
Figure 18. A program initiates a call to LAPI, with embedded up- and down-calls and one lock acquisition and release pair.
Correctness of lock sharing

When a lock is shared between LAPI and a user program, but they do not share any data structures, the correctness of sharing needs to be addressed. Before proceeding, let’s clarify the concepts of atomic operation and critical section. An atomic operation is an operation whose execution steps on data structures must be carried out without interruption from other operations on the same data structures. As an example, when a stack is shared by multiple threads, putting an element onto the stack and updating the stack top must be performed atomically to maintain the integrity of the stack. A critical section is a sequence of atomic and non-atomic operations protected by a lock.

If a critical section consists of multiple atomic operations in a sequence, it is correct to break the sequence into multiple critical sections so that each atomic operation lies entirely in one of the smaller critical sections. For example, there are two correct ways that a thread can push two logically-independent elements onto the stack:

method 1:
- lock(stack)
- push(stack, element1)
- push(stack, element2)
- unlock(stack)

method 2:
- lock(stack)
- push(stack, element1)
- unlock(stack)
- lock(stack)
- push(stack, element2)
- unlock(stack)
The first method is more efficient, but it doesn't mean that the two push operations must be done together in one critical section. It is correct for the second method to release the lock after the first push operation and reacquire the lock before the second push operation. The same analogy can be applied to the meaning of critical section under lock sharing. Even if there is one big critical section protected by the shared lock (analogous to method 1) from the user's view, it is possible that LAPI breaks up the big critical section into multiple smaller ones by releasing and reacquiring the shared lock. Therefore, one requirement is placed upon the user to guarantee the correctness of lock sharing: even if you create a big critical section protected by the shared lock, you must still structure the program so that any of its atomic operations complete before control is transferred to LAPI.

This requirement is useful because:

- LAPI can generate callbacks into your program. If there are any incomplete atomic operations, you must be very careful not to break the atomicity of the incomplete operations in the callbacks.
- It lets your program easily switch from using a shared lock to using a separate lock because using a separate lock requires you to complete atomic operations before the user lock is released and the control is transferred to LAPI.
- LAPI can safely release the shared lock to allow processing in a different thread and callbacks into your program from that thread.

To meet this requirement, you would structure your program so that it functions even if the shared lock is always released before calling LAPI and reacquired after LAPI returns, as illustrated by Program B versus Program A in Figure 20 on page 121 below. Such a thinking process always assumes that LAPI would release the shared lock and reacquire it, which is what LAPI could do to the shared lock.
Implications and restrictions

Initialization and termination

Your program must not hold a shared lock when calling `LAPI_Init` and `LAPI_Term`. In the case of `LAPI_Init`, the lock is not available yet. In the case of `LAPI_Term`, it is not on the performance path and it is better for LAPI to have the lock released before the call.

Figure 20. Critical sections under lock sharing. (a) Program with a large critical section, (b) Program with multiple smaller critical sections, (c) Execution flow of the two programs
Other LAPI calls

For all LAPI calls other than `LAPI_Init` and `LAPI_Term`, LAPI detects whether a shared lock is already held by the running thread. If it is held, LAPI does not acquire the lock again and keeps the lock upon return; otherwise, LAPI acquires the lock and releases it upon return. You can decide whether your program will acquire the shared lock before calling LAPI. For better performance, your program should keep the already-acquired lock before calling LAPI.

Callbacks

The shared lock is guaranteed to be held by the running thread when the following user callbacks are invoked. You should not try to acquire the shared lock and you must not release the lock in these callbacks:

- Send completion handler: invoked when the user header and data buffers for sending a message can be reused.
- Header handler: invoked when the first user header of a message arrives at the receive side.
- Inline completion handler: invoked when a message is completely received, upon the user’s request specified by the header handler.

All of the callbacks can be on the latency path, so avoiding extra locking improves latency.

The out-of-line completion handler is invoked from LAPI’s completion handler thread. It is not on the latency path and the thread will not be holding the shared lock upon the entry of the handler. If your program has any critical sections to process, the shared lock must be acquired first. Calls to LAPI can be made with the lock held.

The error handler will always be invoked without the thread holding the shared lock. Your program should terminate the job upon error, possibly without returning control to LAPI at all.

Long critical sections

Use a shared lock to protect critical sections in communication. You should not use it for long computation, because this could slow down progress in communication. If you want to hold the lock for a long period, your program must poll LAPI by calling `LAPI_Probe` or `LAPI_Msgpoll` in a timely manner so that the communication layer can make progress. Such timely polling is required not only for communication but also for the out-of-line completion handler, checkpoint handler, and error handler to acquire the lock.

Lock preemption

In order to support out-of-line completion, checkpoint/restart, and gang scheduling, in which the handlers must acquire the shared lock first, LAPI will release the lock for the handlers even if it is acquired by your program. The release will only be done when your program calls LAPI. It will never happen that one thread releases a lock acquired by another thread. Because of this, any of your program’s long critical sections must poll LAPI in a timely manner in order to give the handlers a chance to run.

Receive/timer interrupts

Suppose your program initializes using the following sequence:
call LAPI_Init
call LAPI_Util to retrieve the shared lock
lock(shared lock)
initialize global data structure
unlock(shared lock)

A receive/timer interrupt could start before the global data structure is initialized.
Your program must not operate on uninitialized data. There are two ways to be sure of this:
1. Check to see if the data structure has been initialized in callbacks from LAPI.
2. Use a different lock to protect the initialization of data and call LAPI_Init only after the global data structure is initialized.

Performance of multi-threaded programs
Using one shared lock makes locking granularity coarse and there may be more contentions on the lock between threads. In case there is performance degradation, you can change your program’s locking structure to reduce situations in which one thread is blocking another thread. There are two possible ways to do this:
1. A polling thread that is holding the shared lock yields the lock to other threads.
2. Other threads hand off their work to the polling thread.

Compatibility
Existing LAPI programs that don’t exploit lock sharing will run without any change.
To test for the presence of the lock sharing function, you can call LAPI_Util with LAPI_GET_THREAD_FUNC and check for a return code. If any error is returned, lock sharing must not be used.

A sample lock sharing program
A sample lock sharing program follows. This program measures LAPI latency with and without lock sharing. It illustrates the following aspects of lock sharing:
• how to detect whether lock sharing is available
• how to retrieve functions for lock sharing
• how to use the basic lock and unlock functions on the shared lock

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <lapi.h>

#define RC(statement) \ 
{ \ 
int rc = statement; \ 
if (rc != 0) \ 
{ \ 
printf(#statement " rc = %d, line %d\n", rc, __LINE__); \ 
exit(-1); \ 
} \ 
}

lapi_handle_t hndl;
int my_ID;

/* Function to get current time in microseconds */
double microseconds()
struct timeval time_v;
gettimeofday(&time_v, NULL);
return ( (double)time_v.tv_sec * uS_PER_SECOND + time_v.tv_usec );

/* A dummy header handler to consume the incoming message */
void *
am_hndlr(lapi_handle_t *hdl, int *u_hdr, uint *hdr_len,
          lapi_return_info_t *ret_info,
          compl_hndlr_t **chndlr, void **saved_info)
{
    *chndlr = NULL;
    *saved_info = NULL;
    ret_info->ctl_flags = LAPI_BURY_MSG;
    return NULL;
}

/* Function to measure LAPI latency */
double
lapi_latency(int num_pongs, int share_lock)
{
    int rc, i;
    lapi_xfer_t xfer;
    lapi_msg_info_t msg_info;
    double t1, t2;
    lapi_thread_func_t tf;

    /* Retrieve function for lock sharing */
    if (share_lock)
    {
        tf.Util_type = LAPI_GET_THREAD_FUNC;
        rc = LAPI_Util(hndl, (lapi_util_t *)&tf);
        if (rc != LAPI_SUCCESS)
        {
            printf("Lock sharing is not supported by this library.\n");
            share_lock = 0;
        }
    }

    /* Clear structures */
    bzero(&msg_info, sizeof(msg_info));
    bzero(&xfer, sizeof(xfer));

    /* Setup command structure for transfer */
    xfer.Xfer_type=LAPI_AM_XFER;
    xfer.Am.hdr_hdl=1;
    xfer.Am.tgt=1^my_ID;
    xfer.Am.uhdr=NULL;
    xfer.Am.uhdr_len=0;
    xfer.Am.udata=NULL;
    xfer.Am.udata_len=0;
    xfer.Am.shdlr=NULL;
    xfer.Am.sinfo=NULL;
    xfer.Am.tgt_cntr=NULL;
    xfer.Am.org_cntr=NULL;
    xfer.Am.cmpl_cntr=NULL;

    /* Acquire shared lock */
    if (share_lock)
    {
        tf.mutex_lock(hndl);
        RC( LAPI_Gfence(hndl) );
        t1 = microseconds();
    }

    /* Ping-pong test for latency. Count 1 is used in LAPI_Msgpoll
     * to magnify the effect of lock sharing.
if (my_ID == 0) {
  for (i=0; i<num_pongs; i++) {
    /* send then receive */
    RC( LAPI_Xfer(hndl, &xfer) );
    msg_info.status = 0;
    while (!msg_info.status & LAPI_RECV_COMPLETE) {
      RC( LAPI_Msgpoll(hndl, 1, &msg_info) );
    }
  }
} else {
  for (i=0; i<num_pongs; i++) {
    /* Receive then send */
    msg_info.status = 0;
    while (!msg_info.status & LAPI_RECV_COMPLETE) {
      RC( LAPI_Msgpoll(hndl, 1, &msg_info) );
    }
    RC( LAPI_Xfer(hndl, &xfer) );
  }
}

  t2 = microseconds();
  RC( LAPI_Gfence(hndl) );
  RC( LAPI_Gfence(hndl) );
  /* Release shared lock */
  if (share_lock)
    tf.mutex_unlock(hndl);
  /* Calculate latency */
  return (t2 - t1)/num_pongs/2;
}

/*
 * This testcase is to be invoked with 2 optional arguments.
 * 1st arg: number of ping-pongs to measure latency
 * 2nd arg: number of times to measure latency repeatedly.
 */
main(int argc, char *argv[])
{
  share_lock = 1;
  num_pongs = 10000;
  times = 10, i;
  lapi_info_t lapi_info;
  double l0, l1;

  /* Read arguments */
  if (argc > 1)
    num_pongs = atoi(argv[1]);
  if (argc > 2)
    times = atoi(argv[2]);

  /* Improvement of lock sharing is more significant */
  /* when checkpoint is enabled */
  putenv("CHECKPOINT=yes");

  /* Initialize LAPI */
  bzero(&lapi_info, sizeof(lapi_info));
  lapi_info.lib_vers = LAST_LIB;
  RC( LAPI_Init(&hndl, &lapi_info) );

  /* Query/set LAPI settings */
  RC( LAPI_Qenv(hndl, TASK_ID, &my_ID) );
  RC( LAPI_Senv(hndl, INTERRUPT_SET, 0) );
  RC( LAPI_Senv(hndl, ERROR_CHK, 0) );
/* Set address index for header handler */
RC( LAPI_Addr_set(hndl, (void*)&am_hndlr, 1) );

/* Performance latency tests */
for (i=0; i<times; i++) {
    l0 = lapi_latency(num_pongs, 0);
    l1 = lapi_latency(num_pongs, 1);
    if (my_ID == 0)
        printf( "Latency without lock sharing: %.2f us, with: %.2f us\n", 
                l0, l1);
}

/* Terminate LAPI */
RC( LAPI_Term(hndl) );

Use the following command to compile your program:

```
mpcc_r program.c -o program
```

where `program` is the name of your lock sharing program.

The program should be run with two tasks. The output will look like this:

```
Latency without lock sharing: 13.75 us, with: 12.89 us
```
Chapter 15. Striping, failover, and recovery

This chapter describes LAPI's striping, failover, and recovery functions. On systems with multiple switch adapters per node, a job can request the use of multiple LAPI instances for each of the job tasks in order to obtain higher availability and performance in the presence of link/adapter failures. When these multiple instances span multiple physical adapters, LAPI can improve availability of communication when it is used in conjunction with the group services component of RSCT. Using group services, LAPI can quickly determine when an adapter's ability to communicate has ceased, and can then “fail over” all communication to the remaining instances where communication is still possible. Correspondingly, using group services, LAPI is also able to determine when an adapter's ability to communicate has resumed, and can then resume communication using the corresponding instances. During a job run, any failover and recovery operations are, for the most part, transparent to the user.

In this book, adapters that have lost their ability to communicate (as detected by group services), either due to an adapter failure or due to a failure in the network path leading to the adapter, are referred to as “down” adapters. Adapters that still have the ability to communicate are referred to as “up” adapters.

Using failover and recovery

System requirements:

One of the following configurations —

1. The HPS or the InfiniBand switch with RSCT LAPI for AIX, PE for AIX, and TWS LoadLeveler, running over US, with multiple adapters per node
2. The InfiniBand switch and LAPI for Linux, PE for Linux, and TWS LoadLeveler, running over US, with multiple adapter ports per node

All of the nodes in the system must be configured as part of a single RSCT peer domain. Each adapter or adapter port must connect to a separate network. RSCT must be installed and running on the LPAR prior to installation and startup of PNSD.

LAPI's failover and recovery function consists of two basic elements:

1. Monitoring and receiving notification about the communication status of switch adapters.
   - This element depends on the group services component of RSCT and LAPI's protocol network services daemon (PNSD).
2. The use of multiple switch adapters for redundancy, to enable failover.
   - This element depends on TWS LoadLeveler, with corresponding POE functions that serve as a wrapper to convey requests to TWS LoadLeveler.

Failover and recovery cannot be provided for a job if either of these elements is absent.

Monitoring adapter status

Adapter status monitoring depends on the PNSD and the group services component of RSCT.
The NAM component of RSCT LAPI for AIX

For HPS systems, the network availability matrix (NAM) is a pseudo-device component that is packaged as a separate LAPI fileset (rsct.lapi.nam). To make use of LAPI’s failover and recovery function, you can optionally have the NAM pseudo-device “Available” on all of the nodes that are running your job tasks. The pseudo-device nampd0 is automatically created and configured in the boot process after the rsct.lapi.nam fileset is installed. See Chapter 4, “Installing RSCT LAPI for AIX,” on page 29 for more information.

On any particular node, the NAM pseudo-device serves as a repository for status information about the HPS adapters that are within the same RSCT peer domain as that node. The NAM serves as an interface through which the communication status of adapters that are monitored by the group services component can be easily conveyed to LAPI tasks. A change in the communication status of an adapter may reflect that the adapter has gone “down” or has come back “up”. Every time the group services component detects a change in communication status of any of the HPS adapters, it conveys the new status of the changed adapters to the NAM pseudo-device on each of the nodes within the corresponding RSCT peer domain. This, in turn, triggers a notification to LAPI tasks running on those nodes that use multiple instances. On receiving this notification, the LAPI protocol for these tasks can fail over communication from “down” adapters to other “up” adapters. The LAPI protocol can also recover the use of adapters that have again become “up” to resume sharing the communication load.

The PNSD component of LAPI

For non-HPS systems, the protocol network services daemon (PNSD) serves as a repository for NAM and network resource table (NRT) information. For each task in a job, the NRT captures location and resource usage information. Window resources on the adapter are reserved by making NRT API calls. These calls are supported with a separate NRT table API library that interacts with the PNSD. Window resources can be queried and NRTs can be loaded and unloaded using other API calls. The acquiring of window resources and the loading and unloading of NRTs, operations that require root privileges, are typically performed by a resource manager such as TWS LoadLeveler. The NRT is also queried by the running job’s tasks at startup to determine the overall task geometry. See the RSCT: NRT API Programming Guide for more information.

The NAM in LAPI for Linux is similar to the NAM pseudo-device described in “The NAM component of RSCT LAPI for AIX.” It serves as a repository for connection status information that is reported by RSCT group services and as a notification agent for triggering LAPI’s failover and recovery function.

The PNSD is controlled by the system resource controller (SRC), which starts and re-spawns the daemon automatically. You can use SRC commands to control the PNSD. To start the PNSD, enter:

    startsrc -s pnsd

To stop the PNSD, enter:

    stopsrc -s pnsd

The SRC is a part of the AIX operating system. A Linux version of the SRC is shipped with RSCT for Linux. For more information about the SRC, see the appropriate AIX Operating system and device management publication or RSCT: Administration Guide.
RSCT sends network status to the PNSD, which then sends this information to LAPI. All transactions are logged in the PNSD log file: /tmp/serverlog. Older data is stored in the /tmp/serverlog.old file. You need to log in as root to read these files. Using the information in the PNSD log file, you can:
- check the adapter status that is reported by RSCT
- determine which job IDs are running
- determine which resources are allocated

For example, job timeouts or TWS LoadLeveler errors could signal that an adapter is down. To verify this, you can check the PNSD log file to see what RSCT reported about the adapter state.

PNSD can report an unconfigured state for an adapter port if its network ID = 0, its logical ID = 0, and its port status is down. If all three of these conditions are true, the port status will be assigned an unconfigured state.

**PNSD configuration file**

LAPI provides a way for you to configure PNSD that includes specifying a path name for PNSD log files or socket files. You can edit a PNSD configuration file to specify supported PNSD key/value pairs. The root user can create a configuration file with the path name /etc/PNSD.cfg/, with a file permission of 0644. This gives all users read permission. Only root has write permission. If a configuration file is not found on the system at PNSD startup time, PNSD will create this file with default key/value pairs and will use these default values for the PNSD session that is starting up. You can change these values to suit your needs (to avoid using the /tmp directory for log files for security purposes, for example). If changes are made to the PNSD configuration file, PNSD will need to be restarted for the change to take effect. All PNSD users also need to restart their programs in order to connect to the new PNSD session.

Only existing paths are allowed for key values that specify file locations. The file permission of the configuration file (0644), as well as the log files (0600) or socket files (0666), must be strictly maintained. Comments are permitted in the configuration file by preceding each comment line with a number (#) sign. The supported key/values pairs are:

- log_file = /tmp/serverlog
- socket_file = /tmp/PNSD

**Group services and RSCT peer domains**

The group services component updates adapter status in the NAMs or PNSDs of the nodes within a given peer domain. For LAPI failover and recovery to be possible for a given job, job tasks must all run on nodes that belong to the same peer domain. All of the nodes in the system must be configured as part of a single RSCT peer domain. For information about setting up an RSCT peer domain, see RSCT: Administration Guide.

If multiple peer domains are required on an HPS system running AIX, you must make sure that each peer domain forms a separate TWS LoadLeveler resource pool, so jobs that require failover and recovery do not span multiple peer domains. The updating of the NAM pseudo-device by group services is transparent to the user.

InfiniBand systems running Linux require a single, system-wide RSCT peer domain.
**Requesting the use of multiple adapters**

You can use POE environment variables or TWS LoadLeveler job control file (JCF) keywords to request the use of multiple adapters.

**Using POE environment variables**

In order for there to be sufficient redundancy to handle at least one adapter failure, each task of the job needs to be allocated communication instances across at least two different switch adapters. An *instance* is an entity that is required for communication over an adapter device. In US mode, which is specified by setting `MP_EUILIB=us`, an instance corresponds to an adapter window.

Depending on the number of networks in the system and the number of adapters each node has on each of the networks, you can request the allocation of multiple instances for your job tasks by using a combination of the POE environment variables `MP_EUIDEVICE` and `MP_INSTANCES`. The distribution of these requested instances among the various switch adapters on the nodes is done by TWS LoadLeveler. Depending on the resources available on each of the adapters, TWS LoadLeveler will try to allocate these instances on different adapters.

To request the use of multiple instances on a system where all nodes have adapters on each of the \( n \) networks in the system, set `MP_EUIDEVIC=sn_all`. This setting translates to a request for the default number of instances (1) from adapters on each of the networks in the system, and a request for a total of \( n \) instances for each of the job tasks. You do not have to set the `MP_INSTANCES` environment variable. If `MP_EUIDEVICE` is set to `sn_all` and you do set the `MP_INSTANCES` variable to a value \( m \) (where \( m \) is a number from 1 through the value of the case-insensitive string `max`), this translates to a request of \( m \) instances from each of the networks in the system for each job task. For user space, this corresponds to a request for \((m \times n)\) different windows for each job task.

You must take the following considerations into account while defining the number of instances to use:

- If \( m \) is greater than the number of adapters a node has on one of the networks, multiple windows will be allocated from some of the adapters.
- TWS LoadLeveler translates the value `max` as a request to allocate the number of instances (as specified by the `max_protocol_instances` variable) that are defined for this job class in the TWS LoadLeveler `LoadL_admin` file. See *TWS LoadLeveler: Using and Administering* for more information. If you request more instances than the value of `max_protocol_instances`, TWS LoadLeveler allocates a number of instances that is equal to the value of `max_protocol_instances`. To have your job use all adapters on the system across all the networks, you can have the administrator set `max_protocol_instances` for your job class to the number of adapters each node has on each network (assuming that each node has the same number of adapters on each network), and then run your job with `MP_EUIDevice=sn_all` and `MP_INSTANCES=max`.
- On a system where every node is connected to more than one common network, setting `MP_EUIDEVICE=sn_all` is sufficient to allocate instances from distinct adapters for all job tasks. You do not need to set `MP_INSTANCES`. This is because an adapter is connected to exactly one network, this is a request for instances from each network, and if the request is satisfied, at least two distinct adapters have been allocated for each of the job tasks. In the case of user space, if all windows on the adapters of one or more networks are all used up, the job will not be scheduled until windows are available on adapters of each network.
The only guaranteed way to get multiple adapters allocated to the job to satisfy the basic requirements for LAPI’s failover and recovery function, is to have the nodes in the system connect to multiple HPS or IB networks and setting

\[ \text{MP\textunderscore EUDEVICE} = \text{sn\_all}. \]

POE will post an attention message stating that failover and recovery operations may not be possible for the job if multiple instances are requested, but one or more job tasks are allocated instances that are all from the same adapter. The interaction among the values of \text{MP\_INSTANCES}, \text{MP\textunderscore EUDEVICE}, and \text{MP\textunderscore EUILIB}, in terms of the total instances that are allocated to every task of the job, and whether use of the failover and recovery function is possible as a result, are shown in Table 24 and Table 25.

Table 24. Failover and recovery operations on AIX

<table>
<thead>
<tr>
<th>MP\textunderscore EUDEVICE=</th>
<th>Instances allocated per task with MP\textunderscore EUILIB=us</th>
<th>MP_INSTANCES is not set</th>
<th>MP_INSTANCES=m</th>
</tr>
</thead>
<tbody>
<tr>
<td>sn_single</td>
<td>1</td>
<td>no failover</td>
<td>( m )</td>
</tr>
<tr>
<td></td>
<td>no failover</td>
<td></td>
<td>no failover</td>
</tr>
<tr>
<td>sn_all</td>
<td>( \text{num_networks} )</td>
<td>failover is possible if ( \text{num_networks} &gt; 1 )</td>
<td>( m \times \text{num_networks} )</td>
</tr>
<tr>
<td></td>
<td>failover is possible if ( \text{num_networks} &gt; 1 )</td>
<td></td>
<td>failover is possible if ( \text{num_networks} &gt; 1 )</td>
</tr>
</tbody>
</table>

Table 25. Failover and recovery operations on Linux

<table>
<thead>
<tr>
<th>MP_DEVTYPE=ib and MP\textunderscore EUDEVICE=</th>
<th>Instances allocated per task with MP\textunderscore EUILIB=us</th>
<th>MP_INSTANCES is not set</th>
<th>MP_INSTANCES=m</th>
</tr>
</thead>
<tbody>
<tr>
<td>sn_single</td>
<td>1</td>
<td>no failover</td>
<td>( m )</td>
</tr>
<tr>
<td></td>
<td>no failover</td>
<td></td>
<td>no failover</td>
</tr>
<tr>
<td>sn_all</td>
<td>( \text{num_networks} )</td>
<td>failover is possible if ( \text{num_networks} &gt; 1 )</td>
<td>( m \times \text{num_networks} )</td>
</tr>
<tr>
<td></td>
<td>failover is possible if ( \text{num_networks} &gt; 1 )</td>
<td></td>
<td>failover is possible if ( \text{num_networks} &gt; 1 )</td>
</tr>
</tbody>
</table>

Using TWS LoadLeveler JCF keywords

The use of the TWS LoadLeveler job class attribute \texttt{max\_protocol\_instances} is described in “Using POE environment variables” on page 130. For more information about this attribute, and for the syntax to specify the request for multiple instances on a single network or on all networks in the system using a TWS LoadLeveler job control file (JCF), see TWS LoadLeveler: Using and Administering.

Note: Although more than eight instances are allowed using a combination of TWS LoadLeveler’s \texttt{max\_protocol\_instances} setting and the \texttt{MP\_INSTANCES} environment variable, LAPI ignores all window allocations beyond the first eight, because LAPI supports a maximum of eight adapters per operating system instance and the best performance can be obtained with one window on each of them. Using multiple windows on a given adapter provides no performance advantage.

Failover and recovery restrictions

- When a job with a failed adapter is preempted, TWS LoadLeveler may not be able to continue with the job if it (TWS LoadLeveler) cannot reload the switch
table on the failed adapter. Any adapter failure that causes switch tables to be
unloaded will not be recovered during the job run.

- In single-network scenarios, TWS LoadLeveler tries to allocate adapter windows
  on separate adapters, but does not always succeed. Correspondingly, failover
  and recovery are not always possible in single-network scenarios. The user will
  get POE attention messages at job startup time when TWS LoadLeveler fails to
  get windows on at least two separate adapters.
- On Linux, failover and recovery functions are supported only for multi-network
  configurations.
- Failover and recovery functions are not supported on standalone systems.

### Data striping

When running parallel jobs on HPS or InfiniBand systems, it is possible to stripe
data through multiple adapter windows. This is supported for both IP and US
protocols.

If the system has more than one switch network, the resource manager allocates
adapter windows from multiple adapters. A *switch network* is the circuit of adapters
that connect to the same switch. One window is assigned to an adapter, with one
adapter each selected from a different switch network.

If the system has only one switch network, the adapter windows are most likely
allocated from different adapters, provided that there are sufficient windows
available on each adapter. If there are not enough windows available on one of the
adapters, the adapter windows may all be allocated from a single adapter.

LAPI manages communication among multiple adapter windows. Using resources
that LoadLeveler allocates, LAPI opens multiple user space windows for
communication. Every task of the job opens the same number of user space
windows, and a particular window on a task can only communicate with the
corresponding window on other tasks. These windows form a set of "virtual
networks", in which each "virtual network" consists of a window from each task that
can communicate with the corresponding windows from the other tasks. The
distribution of data among the various windows on a task is referred to as *striping*,
which has the potential to improve communication bandwidth performance for LAPI
clients.

To enable striping in user space (US) mode, use environment variable settings that
result in the allocation of multiple instances. For a multi-network system, this can be
done by setting `MP_EUIDEVICE` to `sn_all`. On a single-network system with
multiple adapters per operating system image, this can be done by setting
`MP_EUIDEVICE` to `sn_single` and `MP_INSTANCES` to a value that is greater than
1. For example, on a node with two adapter links, in a configuration where each link
is part of a separate network, the result of setting `MP_EUIDEVICE` to `sn_all` is a
window on each of the two networks, which are independent paths from one node
to others. See "[Requesting the use of multiple adapters](#)" on page 130 for more
information.

For IP communication and for messages that use the user space FIFO mechanism
(in which LAPI creates packets and copies them to the user space FIFOs for
transmission), striping provides no performance improvement. Therefore, LAPI does
not perform striping for short messages, non-contiguous messages, and all
communication in which bulk transfer is disabled through environment variable
settings.
For large contiguous messages that use bulk transfer, striping provides a vast improvement in communication performance. Bandwidth scaling is nearly linear with the number of adapters (up to a limit of eight) for sufficiently-large messages. This improvement in communication bandwidth stems from: 1) the low overhead that is needed to initiate the remote direct memory access (RDMA) operations used to facilitate the bulk transfer, 2) the major proportion of RDMA work that is being done by the adapters, and 3) high levels of concurrency in the RDMA operations for various parts of the contiguous message that are being transferred by RDMA by each of the adapters. For more information about RDMA, see "Bulk message transfer on AIX" on page 63.

To activate striping or failover for an interactive parallel job, you must set the MP_EUIDEVICE and MP_INSTANCES environment variables as follows:

- For instances from multiple networks:
  
  MP_EUIDEVICE=sn_all — Guarantees that the adapters assigned will be from different networks.

- For instances from a single network:
  
  MP_EUIDEVICE=sn_single and MP_INSTANCES=n (where n is greater than 1 and less than max_protocol_instances) — Improved striping performance using RDMA can only be seen if windows are allocated from multiple adapters on the single network. Such an allocation may not be possible if there is only one adapter on the network or if there are multiple adapters, but there are available resources on only one of the adapters.

To activate striping for a parallel job submitted to the LoadLeveler batch system, the network statement of the LoadLeveler command file must be coded accordingly.

- Use this network statement for a LAPI US job that uses High Performance Switches on multiple networks:
  
  `# network.lapi = sn_all,shared,us`

- Use this network statement for an MPI and LAPI US job that uses High Performance Switches on multiple networks and shares adapter windows:
  
  `# network.mpi_lapi = sn_all,shared,us`

The value of MP_INSTANCES ranges from 1 to the maximum value specified by max_protocol_instances, as defined in the LoadLeveler LoadL_admin file. The default value of max_protocol_instances is 1. See TWS LoadLeveler: Using and Administering for more information.

### Communication and memory considerations for AIX

Depending on the mode of communication, when multiple HPS adapters are used for data striping or for failover and recovery, additional memory or address space resources are used for data structures that are associated with each communication instance. In 32-bit applications, these additional requirements have implications that you must consider before deciding whether to use striping or failover and recovery and the extent to which you will use these functions.

#### IP communication

- When multiple HPS instances are used for IP communication, RSCT LAPI for AIX allocates these data structures from the user heap. Some 32-bit applications may therefore need to be recompiled to use additional data segments for their heap by using the -bmaxdata loader flag and requesting a larger number of segments. The default amount of data that can be allocated for 64-bit programs is practically unlimited, so no changes are needed. Alternatively, you can modify the 32-bit
executable using the AIX ldedit command or by setting the LDR_CNTRL environment variable to MAXDATA. Base the increase to -bmaxdata on what is needed rather than setting it to the maximum allowed (0x80000000). Using more segments than required may make certain shared memory features unusable, which can result in poor performance. Also, applications that require the eight allowed segments for their own user data (thus leaving no space for LAPI to allocate structures) must use a single IP instance only (MP_EUIDEVICE=sn_single).

For more information about ldedit, see AIX 5L Version 5.3 Commands Reference or AIX 5L Version 6.1 Commands Reference. For more information about LDR_CNTRL, see AIX 5L Version 5.3: Performance Management Guide or AIX 5L Version 6.1: Performance Management Guide.

- When multiple adapters from each of one or more networks are used for IP communication, with multiple adapters in each IP subnet, failover can occur only if the AIX IP routing table is updated appropriately to remove the failed adapters. This is because: 1) AIX’s IP routing alternates communication to a given target IP address among all the routes to that target in its routing table, 2) with multiple adapters on each IP subnet, multiple routes may be detected and stored in the IP routing table for each remote adapter that is on the same subnet and, 3) the loss of an adapter does not automatically result in an update of the IP routing table. Without such an update to the routing table, the underlying route used to reach a given target IP address may still go through a failed adapter connection internally, thus preventing proper communication even if LAPI has failed over to a UDP socket that leads to a target adapter that is detected to be still functioning well. This problem does not affect the case where the multiple adapters on each node are organized one each on separate IP subnets. See AIX 5L Version 5.3: System Management Guide, Communications and Networks or AIX 5L Version 6.1: System Management Guide, Communications and Networks for more information.

US communication

When multiple HPS instances are used for US communication, you need to consider the following segment usage information when deciding whether to use striping or failover and recovery. The communication subsystem uses segment registers for several different purposes. The AIX memory model for 32-bit applications uses five segment registers. In a 32-bit executable, there are only 16 segment registers available. In a 64-bit executable, the number of segment registers is essentially unbounded. Because segment registers are abundant in 64-bit job runs, this discussion is important only for 32-bit job runs.

By default, the amount of memory that is available for application data structures (the “heap”) in a 32-bit job run is somewhat less than 256MB. You can use the -bmaxdata:0x80000000 loader flag to allocate 2GB of heap space, but this requires eight segment registers. Smaller -bmaxdata values use fewer segment registers, but these values limit the size of application data structures. If you try to use every available feature of the communication subsystem and allow 2GB for the heap, there will not be enough registers, and your application will lose some performance or perhaps not be able to start. The communication subsystem uses segments as follows:

- One US instance (window): 2
- Each additional instance: 1
- Switch clock: 1 (applies only to MPI when it is used in conjunction with RSCT LAPI for AIX on the HPS)
- Shared memory: 1
• Shared memory cross-memory attach: 1

Using MPI and RSCT LAPI for AIX together with separate windows consumes segments beyond the minimum. Using striping also consumes extra windows. When MPI is used in conjunction with LAPI, access to the switch clock by MPI for the MPI_WTIME_IS_GLOBAL attribute requires a dedicated segment register on the HPS. Turning shared memory communication on requires one segment register for basic functions and a second segment register to exploit cross-memory attach, to accelerate large messages between tasks on the same node. If your application requires a large heap, you may need to forgo some communication subsystem options.

For most MPI applications, if you are using RSCT LAPI for AIX and the HPS, you can set MP_CLOCK_SOURCE=AIX and free one register. If MPI and RSCT LAPI for AIX calls are used in the application, make sure MP_MSG_API is set to MPI_LAPI rather than MPI,LAPI. Because shared memory uses one pair of registers per protocol, using MPI_LAPI rather than MPI,LAPI is especially important when combining shared memory and user space. If you do not need to use the striping and failover functions, make sure that MP_EUIDevice is set to sn_single and that MP_INSTANCES is set to 1 or is not set (in which case, it defaults to 1).

For 32-bit executables that are compiled to use small pages, the segment registers that are reserved by AIX and by -bmaxdata are claimed first. The initialization of user space comes second. If there are not enough registers left, your job will not start. The initialization of shared memory comes last. If there are no registers left, the job will still run, but without shared memory. If there is only one register left, shared memory will be enabled, but the optimization to speed large messages with cross-memory attach will not be used. If there are no registers left, shared memory will be bypassed and on-node communication will go through the network.

For 32-bit executables that use large pages, dynamic segment allocation (DSA) is turned on automatically, so any -bmaxdata segments requested are not reserved first for the user heap, but are instead allocated in the order of usage. Thus, if the program allocates memory corresponding to the total size of the requested -bmaxdata segments before MPI_Init or LAPI_Init is called, the behavior would be similar to the small page behavior that is described in the previous paragraph. However, if MPI_Init or LAPI_Init is called before the memory allocation, segments that were intended for use for the program heap may be first obtained and reserved for windows and for communication library features such as shared memory. In this case, the program will be left with fewer segments to grow the heap than -bmaxdata had requested. The program is likely to start by claiming all the segments required for the initialization of the communication subsystem, but will terminate later in the job run on a malloc failure as its data structure allocations grow to fill the space that the specified -bmaxdata value was expected to provide.


For information about DSA, see AIX 5L Version 5.3: General Programming Concepts, Writing and Debugging Programs or AIX 5L Version 6.1: General Programming Concepts, Writing and Debugging Programs.
Chapter 16. Threaded programming

General guidelines

LAPI has no concept of identifying individual threads within a task. No communication can be directed to a specific thread.

In multi-threaded programming, synchronization among threads is the user's responsibility. All LAPI communication is based on the use of handles. If multiple threads share the same handle, their calls to LAPI are serialized in the LAPI library, which makes LAPI safe for multi-threaded programming.

It is recommended that you don't write threaded message-passing programs until you are very familiar with writing and debugging single-threaded multitask programs and multithreaded single-task programs.

Using LAPI_Address_init

Here is an example of using `LAPI_Address_init` in multiple threads. Suppose there are two tasks creating two threads to do the exchange of addresses as follows.

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Task 1</th>
<th>Thread 2</th>
<th>Task 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LAPI_Address_init(hndl, addr1, addr_tab1);</td>
<td>Task1</td>
<td>LAPI_Address_init(hndl, addr1, addr_tab1);</td>
</tr>
<tr>
<td></td>
<td>LAPI_Address_init(hndl, addr2, addr_tab2);</td>
<td>Task2</td>
<td>LAPI_Address_init(hndl, addr2, addr_tab2);</td>
</tr>
</tbody>
</table>

The user expects that all `addr1` will be collected in `addr_tab1` and all `addr2` will be collected in `addr_tab2`. However, if `Thread 1` and `Thread 2` are running completely in parallel for this code section, the resulting serialization in LAPI can be any one of the following sequences.

<table>
<thead>
<tr>
<th>Sequence 1</th>
<th>Sequence 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPI_Address_init(hndl, addr1, addr_tab1);</td>
<td>LAPI_Address_init(hndl, addr2, addr_tab2);</td>
</tr>
<tr>
<td>LAPI_Address_init(hndl, addr2, addr_tab2);</td>
<td>LAPI_Address_init(hndl, addr1, addr_tab1);</td>
</tr>
</tbody>
</table>

Suppose Task 1 is serialized into Sequence 1 and Task 2 into Sequence 2. The result of address exchange in this scenario will be unexpected because `addr1` of Task 1 will be put into `addr_tab2` of Task 2 and `addr2` of Task 2 will be put into `addr_tab1` of Task 1. To achieve the desired results in `addr_tab1` and `addr_tab2`, you can either put the two `LAPI_Address_init` calls in one thread or use your own thread synchronization mechanism to enforce ordering of the two `LAPI_Address_init` calls. For example, the following code allows threads to exchange counter addresses group by group.

```c
volatile int turn = 0;
void *communication_thread(void *param)
{
    int group_id = (int)param;
    lapi_cntr_t tgt_cntr;
    void ***tgt_cntr_tab;
    tgt_cntr_tab = malloc(num_tasks * sizeof(void *));
    while (turn != group_id) yield();
    LAPI_Address_init(hndl, &tgt_cntr, tgt_cntr_tab);
    turn++;
}
```
Making global fence calls

Synchronization of one set of threads across all tasks of the job concurrently with synchronization of another set of threads across all the same tasks cannot be achieved by making one `LAPI_Gfence` call in each thread. This is because, just as with the example of `LAPI_Address_init` above, each task will locally serialize the `LAPI_Gfence` call in undefined order, and it is unpredictable whether the `LAPI_Gfence` call for thread A on task 0 will match the `LAPI_Gfence` call for thread A or thread B on task 1. If you intend to make two global fence calls in two different threads, you should enforce the ordering of the two calls similar to the example of `LAPI_Address_init` ordering above.

Making “wait on counter” calls

Multiple `LAPI_Waitcntr` calls can be issued from different threads without one blocking the others. Note, however, that two threads cannot wait on the same counter. `LAPI_Waitcntr` will return to the user as soon as the counter reaches or exceeds the value that is being waited on, which also implies that an earlier `LAPI_Waitcntr` may return after a later `LAPI_Waitcntr`.

Synchronizing threads across tasks

To synchronize threads in different tasks, you may not want to call `LAPI_Gfence` because it blocks other threads in the same task from any further communication before the global fence operation is complete. A better way to implement such synchronization uses other LAPI functions. The following steps show a possible implementation:

1. Pick a root thread from the group of threads to synchronize.
2. All threads in the group send a barrier message to the root thread task (`LAPI_Amsend, LAPI_Put`) with an associated target counter.
3. All threads wait for a response by waiting on a specific counter to be incremented (`LAPI_Waitcntr`).
4. The root thread waits for its counter to reach a value that is equal to the number of threads in this group (`LAPI_Waitcntr`).
5. The root thread then broadcasts a message to all of the threads in the group specifying the counter they are waiting on as the target counter, to release them from the barrier (`LAPI_Amsend, LAPI_Put`).

Using handlers

Typically, LAPI invokes the following user-provided handlers to indicate the arrival/completion of a message:

- Header handler: when data first arrives at the receiving side in an active message.
- Send completion handler: when data has been sent at the sending side and the user can modify the data buffer.
- (Receive) Completion handler: when data has been completely received at the receiving side.

LAPI does not guarantee which thread will invoke the above handlers, with the exception of non-inline completion handlers, which are always invoked in the completion handler thread created by LAPI. All other types of handlers can be
invoked in any thread where LAPI communication functions are called and also in the interrupt thread. Refer to the next section for a description of the threads in an executing LAPI program.

**LAPI threads**

A program running LAPI is inherently multi-threaded, even though the user program may itself be single-threaded. The list of threads in a program running with LAPI is as follows:

- **User threads**: created by the user.
- **Interrupt thread**: When interrupt is enabled and there are incoming packets from the adapter, LAPI interrupt handler is called in this thread to process packets. All user-provided handlers except non-inline completion handlers can be invoked in this thread.
- **Completion handler threads**: LAPI creates these threads to run non-inline completion handlers.
- **Shared memory dispatcher thread**: If shared memory is on, LAPI creates this thread to handle interrupt in shared memory transport. All user-provided handlers except non-inline completion handlers can be invoked in this thread.

A single-threaded LAPI user program has one user thread, but may have one thread of each of the other types that are transparent to the user.
Chapter 17. Using LAPI on a standalone system

System requirements
One of the following:
• LAPI for Linux over UDP/IP
• RSCT LAPI for AIX

LAPI cannot be run standalone on InfiniBand systems running over US.

A standalone system refers to a system environment that does not include Parallel Environment. Depending on whether you want to run standalone programs over UDP/IP (on AIX or Linux) or US (on AIX), refer to the appropriate README file:

• On AIX:
  /opt/rsct/lapi/samples/standalone/udp/README.LAPI.STANDALONE.UDP
  /opt/rsct/lapi/samples/standalone/us/README.LAPI.STANDALONE.US

• On Linux:
  /opt/ibmhpc/lapi/samples/standalone/udp/README.LAPI.STANDALONE.UDP

Setting up a standalone system
On a standalone system, you need to assign task IDs and initialize jobs on each node. To use LAPI on a standalone system, set the following environment variables:

**MP_CHILD**  Sets the task ID of the current job. MP_CHILD needs to be set to a unique value for each task in standalone mode.

**MP_LAPI_INET_ADDR**
Describes the network setup for IP communication among LAPI tasks. The format is:

```
MP_LAPI_INET_ADDR=@num_instances:IP_addr_in_dotted_decimal,adapter_name[:IP_addr_in_dotted_decimal2,adapter_name2...]
```

where: num_instances is the number of network instances, IP_addr_in_dotted_decimal is the IP address of the adapter being used, and adapter_name is the logical name of the adapter device. The number of IP_addr_in_dotted_decimal,adapter_name pairs is equal to the value of num_instances. For example, if you want a task to use two separate adapters, sn0 and sn1, for IP communication, you would use something like this:

```
MP_LAPI_INET_ADDR=02:192.161.0.1,sn0:192.161.1.1,sn1
```

**MP_LAPI_NETWORK**
Describes the network setup for user space communication among RSCT LAPI for AIX 5L tasks. The format is:

```
MP_LAPI_NETWORK=@num_instances:window_num,adapter_name[:window_num2,adapter_name2...]
```

where: num_instances is the number of network instances and window_num and adapter_name refer to one or more adapter windows that have been reserved for this task. The number of window_num,adapter_name pairs is equal to the value of num_instances. For example, if you loaded one adapter instance for this task using window 164 on adapter sn0, you would use:

```
MP_LAPI_NETWORK=01:164,sn0
```

**MP_PARTITION**
A number that is the same for all tasks in the job. In standalone mode, you need to set this variable to an identical value for each task. In standalone mode for switched communication, the **MP_PARTITION** value must be associated with the network table description file.

**MP_PROCS**
The value of `num_tasks`, which is the total number of program tasks in the job. This number must be the same for all tasks.

See "Variables for standalone systems" on page 324 for more information.

To use LAPI shared memory on a standalone system, set the following environment variables:

**LAPI_USE_SHM**
Enables or disables the use of shared memory.

- **no** -- disables the use of shared memory (the default).
- **yes** -- enables the use of shared memory where it is possible. LAPI will communicate using shared memory among all common tasks (tasks that are on the same node) over the selected device (user space over switch, IP over switch, or IP over Ethernet). See **MP_EUIDEVICE** and **MP_EUILIB** for tasks on different nodes. Shared memory requires segment registers, which can affect availability to user code in 32-bit applications.
- **only** -- communicates only using shared memory. LAPI will fail to initialize if this option is chosen and tasks are assigned to more than one node.

**MP_COMMON_TASKS**
Is set for shared memory jobs. It is different for each task, and is mapped to the setting of the **MP_CHILD** environment variable. For each task, **MP_COMMON_TASKS** contains a string that indicates the number and task IDs of other tasks on the same node (that is, those that can communicate through shared memory).

For **MP_COMMON_TASKS**, the format of the string is:

**MP_COMMON_TASKS=number-of-common-tasks:common-task:other-task-2:...**

For example, for task 1 of a 4-task job running on the same node, the following environment value is set for task 1: **MP_COMMON_TASKS=3:0:2:3**. Notice that task 1 is not in the list. Task numbering starts at 0 to `ntask - 1`, where `ntask` is the total number of tasks.

See "Variables for shared memory" on page 323 for more information.
Note

The descriptions and formats of **MP_COMMON_TASKS**, **MP_LAPI_INET_ADDR**, **MP_LAPI_NETWORK** are provided in this book for informational purposes only. These environment variables are not intended to be used as external programming interfaces. IBM will not guarantee that the formats or values of these variables can continue to be used without change in future releases. Programmers and users who choose to develop applications that depend on these variables do so with the understanding that these variables may be subject to future change. IBM cannot guarantee that such applications can migrate or coexist with future releases without additional changes, nor will IBM ensure that there will be binary compatibility of these variables.

Initializing a standalone system

See "LAPI_Init" on page 199 for examples of standalone initialization.

In UDP/IP mode

For standalone UDP/IP initialization, LAPI must have at least one of these two means of transferring UDP information: a user handler or a user list. If both a user handler and a user list are passed in, the user handler is invoked and the user list is ignored.

In UDP/IP mode, LAPI uses a pair of connectionless sockets for each task, one for reading and one for writing. During initialization, the IP address and port information for each task's read socket must be distributed to all tasks. On a standalone system, you need to distribute this read socket information to all of the tasks. LAPI provides two mechanisms for distributing IP address and port information on standalone systems: user handlers and user lists.

For the user handler mechanism, a user handler is passed to LAPI as a callback pointer to be used during initialization. Before opening the UDP sockets during initialization (in **LAPI_Init**), LAPI calls the handler and expects it to return (by way of a reference parameter) a list of IP address and port information for each task in the job.

For the user list mechanism, you need to pass a pointer to LAPI at initialization time (by way of the **lapi_info_t** structure that is passed in to **LAPI_Init**) that points to user memory that has the required port information.

In US mode on AIX

For standalone initialization in user space (US), no changes are required to C source code that makes LAPI calls, except that the code must be compiled with a non-parallel compiler (**cc_r**, for example). When running an RSCT LAPI for AIX program standalone over US, you need to handle a number of tasks that are normally handled by PE and TWS LoadLeveler. In particular, you need to:

* Determine available adapters and windows, and then load the network tables for the desired network configuration on the chosen adapters and windows.

* Set the **MP_LAPI_NETWORK** environment variable in the following format:

  ```
  MP_LAPI_NETWORK=@num_instances:window_num,snX
  ```
where \texttt{num\_instances} is the number of network instances to use (this must be set to 1), \texttt{window\_num} is the window on the adapter that has been loaded for this task and \texttt{snX} is the name of the adapter that has this window and on which the network table has been loaded. For example, if you loaded one adapter instance for this task using window 164 on adapter \texttt{sn0}, you would set

\texttt{MP\_LAPI\_NETWORK} as follows:

\begin{verbatim}
MP_LAPI_NETWORK=01:164,sn0
\end{verbatim}

- Set other environment variables to control various aspects of the LAPI run time. See "Environment variables" on page 317 for more information.
- Invoke each of the tasks separately on the desired set of nodes on which the job is to be run.

For full details on standalone initialization in user space, see the \texttt{README.LAPI.STANDALONE.US} file.

### Compiling LAPI programs on a standalone system

Table 26 and Table 27 show which commands to enter to compile a LAPI program on a standalone system.

#### Table 26. Compiling RSCT LAPI for AIX programs on a standalone system

<table>
<thead>
<tr>
<th>To compile a C program:</th>
<th>\texttt{cc_r program.c -o program}</th>
</tr>
</thead>
<tbody>
<tr>
<td>To compile a C++ program:</td>
<td>\texttt{CC_r program.C -o program}</td>
</tr>
<tr>
<td>To compile a FORTRAN program:</td>
<td>\texttt{xtf_r program.f -o program}</td>
</tr>
</tbody>
</table>

#### Table 27. Compiling LAPI for Linux programs on a standalone system

<table>
<thead>
<tr>
<th>To compile a C program:</th>
<th>\texttt{gcc program.c -o program}</th>
</tr>
</thead>
<tbody>
<tr>
<td>To compile a C++ program:</td>
<td>\texttt{gcc program.C -o program}</td>
</tr>
<tr>
<td>To compile a FORTRAN program:</td>
<td>\texttt{gcc program.f -o program}</td>
</tr>
</tbody>
</table>

### Standalone systems: restrictions

Failover and recovery functions are not supported on standalone systems.

InfiniBand systems running over US cannot be run standalone.
Chapter 18. LAPI commands
pnsd_stat

Purpose
Queries the statistics of user space tasks.

Synopsis
pnsd_stat [-c] [-n] [-q] { [ -j job_ID ... ] | [ -r job_ID ... ] | [ -t task_key ... ] | [ -u user ... ] } [-h]

Description
The pnsd_stat command queries the statistics of user space tasks. When pnsd_stat is run on a computation node, it gives only the statistics of the node.

Use the -u option to query accumulated statistics for specified users. Use the -j option to query accumulated statistics for specified jobs. Use the -r option to query the run-time statistics of the tasks of the specified jobs. The jobs must be running when the pnsd_stat command is issued. Use the -n option to query node-wide statistics accumulated from all previously-finished tasks.

Use the -c option to clean up statistics. When used in conjunction with one or more of these options — -j, -n, -t, or -u — the -c option cleans up the specified statistics.

Use the -q option to list all users and job IDs that have statistics.

The following end-user statistics are related to user space traffic volume:
- Total incoming packets and data size
- Total outgoing packets and data size
- Total RDMA messages and data size

The following end-user statistics are related to failures:
- Retransmitted packet count
- Duplicate packet count
- Number of RDMA messages failing over to FIFO mode
- Number of registration failures
- Number of connection failures

Options
- -c Cleans up the specified statistics. You need root permission to use this option.
- -n Includes node-wide statistics.
- -q Lists all users and job IDs that have statistics.
- -j job_ID ... Includes statistics of the specified jobs.
- -r job_ID ... Includes statistics of the specified jobs at run-time.
pnsd_stat

- `t task_key ...`
  Includes statistics of the specified tasks.

- `u user ...`
  Includes statistics of the specified users. You need `root` permission to query other users’ statistics.

- `h`
  Writes the command’s usage statement to standard output.

Environment

`MP_INFOLEVEL`
Indicates the level of message reporting. Any value that is greater than or equal to 2 causes LAPI to print out such library information as the LAPI version number and the build timestamp.

Standard output

When the `-h` flag is specified, this command’s usage statement is written to standard output.

Standard error

At the beginning of a job or upon the restart of a checkpointed job, task 0 prints the following line to standard error when `MP_INFOLEVEL` is set to 1 or greater:

```
LAPI job ID for this job is: 123409420
```

Because querying statistics (and later invoking triggers) requires this ID, the user needs to set `MP_INFOLEVEL` to at least 1 to use the statistics and trigger features effectively. The job ID here is defined by LAPI and is different from TWS LoadLeveler’s job step ID in concept.

A task key is printed to standard error with the following message when LAPI terminates and `MP_INFOLEVEL` is 2 or greater:

```
Communication statistics of task 29 is associated with task key: 123409420_29
```

The user can use the task key to retrieve the statistics of this task. The task key is the job ID (the string before `_`) followed by the task ID. The user can also use the job ID to query the statistics of the job.

Security

You need `root` permission to use the `-c` option. You need `root` permission to use the `-u` option for querying other users’ statistics.

See also

Commands: pnsd_trigger
The `pnsd_trigger` command calls a trigger. The job ID, which is defined by LAPI, is the same one that is used for querying user space statistics. Only the owner of the job is allowed to invoke triggers within the job. When the `pnsd_trigger` command is issued on a node, all tasks that belong to the specified job on the node will invoke the specified trigger.

LAPI provides the following predefined triggers:

- **lapi_dump_objects** `[file_name]`
  This trigger dumps LAPI's internal objects for debugging purposes.

- **lapi_dump_stat** `[file_name]`
  This trigger displays LAPI's communication statistics.

- **lapi_list_triggers** `[file_name]`
  This trigger lists all of the defined triggers (including LAPI's and the user's).

The optional file name specifies where the output should be directed. Each task outputs to its own file, which is named with this format: `file_name.task_ID`. If `file_name` is not specified, the output is sent to standard error.

**Parameters**

- `trigger_name`
  Specifies the name of the trigger.

- `argument...`
  Is a space-separated list of arguments for the trigger.

**Options**

- `-d job_ID`
  Dumps the contents of LAPI objects for the specified job ID.

- `-j job_ID`
  Specifies the target job ID to invoke the trigger.

- `-l job_ID`
  Lists all registered triggers for the specified job ID.

- `-t timeout`
  Specifies the timeout in seconds for the trigger. The default value is 60 seconds.

- `-h`
  Writes the command's usage statement to standard output.
pnsd_trigger

-v  Writes the command's verbose messages to standard output.

Environment

MP_INFOLEVEL
Indicates the level of message reporting. Any value that is greater than or equal to 2 causes LAPI to print out such library information as the LAPI version number and the build timestamp.

Standard output

When the -h flag is specified, this command's usage statement is written to standard output. All verbose messages are written to standard output.

Standard error

At the beginning of a job or upon the restart of a checkpointed job, task 0 prints the following line to standard error when MP_INFOLEVEL is set to 1 or greater:

LAPI job ID for this job is: 123409420

See also

Commands: pnsd_stat
pnsd_trigger
Chapter 19. A sample LAPI subroutine

For each subroutine in Chapter 20, “Subroutines for all systems (PE and standalone),” on page 157 and Chapter 21, “Subroutines for standalone systems,” on page 281, information about some or all of the following topics is included, as appropriate: purpose, library, C syntax, FORTRAN syntax, parameters, description, restrictions, return values, location, C examples, FORTRAN examples, and related information. Review “lapi_subroutines” on page 154 before proceeding to get a better understanding of how the subroutine information in Chapter 20, “Subroutines for all systems (PE and standalone),” on page 157 and Chapter 21, “Subroutines for standalone systems,” on page 281 is structured.
lapi_subroutines

Purpose

Provides overview information about LAPI subroutines, including some sample sections of the man pages for these subroutines.

Library

Availability Library (liblapi_r.a)

C syntax

```c
#include <lapi.h>

int lapi_subroutines(parm1, parm2...)
    type1 parm1;
    type2 parm2;
    ...
```

FORTRAN syntax

```fortran
include 'lapif.h'

lapi_subroutines(parm1, parm2..., ierror)
    type1 :: parm1;
    type2 :: parm2;
    ...
    integer ierror
```

Parameters

Parameter definitions are listed as follows:

**Input**

- `parm1` Describes `parm1`.

**Input/output**

This section includes all LAPI counters.

- `parm2` Describes `parm2`.

**Output**

Function calls are nonblocking, so counter behavior is asynchronous with respect to the function call.

- `ierror` Specifies a FORTRAN return code. This is always the last parameter.

Description

This man page provides overview information about LAPI subroutines, including some sample sections of the man pages for these subroutines.

**Programming with C++**

LAPI subroutines provide `extern "C"` declarations for C++ programming.

**Programming with FORTRAN**

When writing FORTRAN programs, keep in mind that the FORTRAN programming language is *not* case-sensitive.

**Profiling**

Querying runtime values

You can find out the size (or size range) of certain parameters by calling the 
LAPI_Qenv subroutine with the appropriate query type. For example, call 
LAPI_Qenv with the LOC_ADDRTBL_SZ query type to find out the size of the 
address table used by the LAPI.Addr_set subroutine:

    LAPI_Qenv(hndl, LOC_ADDRTBL_SZ, ret_val)

Now, suppose you want to register a function address using LAPI.Addr_set:

    LAPI.Addr_set (hndl, addr, addr_hndl)

The value of index addr_hndl must be in the range:

    1 <= addr_hndl < LOC_ADDRTBL_SZ

When used to show the size of a parameter, a comparison of values, or a range of 
values, valid values for the query parameter of the LAPI_Qenv subroutine appear 
in **bold, underlined** capital letters. For example:

**NUM_TASKS**

is a shorthand notation for:

    LAPI_Qenv(hndl, NUM_TASKS, ret_val)

See “LAPI_Qenv” on page 221 for a list of the query parameter’s valid values.

Restrictions

Any specific restrictions for the subroutine appear here.

Also, see Appendix H, “LAPI restrictions,” on page 335 for more information.

Return values

**LAPI_SUCCESS**

Indicates that the function call completed successfully.

Any other return values for the subroutine appear here.

For a complete list, see “LAPI return values” on page 313.

For information about LAPI error messages, see RSCT: Messages.

Location

/usr/lib/liblapi_r.a
lapi_subroutines

C examples

Any C examples of the subroutine appear here.

FORTRAN examples

Any FORTRAN examples of the subroutine appear here.

Related information

Any information that is related to the subroutine (including names of related subroutines) appears here.
Chapter 20. Subroutines for all systems (PE and standalone)

Use the subroutines in this chapter on systems that are running Parallel Environment (PE) and on standalone systems.
LAPI_Addr_get

Purpose
Retrieves a function address that was previously registered using LAPI_Addr_set.

Library
Availability Library (liblapi_r.a)

C syntax
#include <lapi.h>

int LAPI_Addr_get(hndl, addr, addr_hndl)

lapi_handle_t hndl;
void **addr;
int addr_hndl;

FORTRAN syntax
include 'lapif.h'

lapi_addr_get(hndl, addr, addr_hndl, ierror)

integer hndl
integer (kind=LAPI_ADDR_TYPE) :: addr
integer addr_hndl
integer ierror

Parameters

Input

hndl    Specifies the LAPI handle.
addr_hndl  Specifies the index of the function address to retrieve. You should have previously registered the address at this index using LAPI_Addr_set. The value of this parameter must be in the range 1 <= addr_hndl < LOC_ADDRTBL_SZ.

Output

addr    Returns a function address that the user registered with LAPI.
ierror    Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: local address manipulation

Use this subroutine to get the pointer that was previously registered with LAPI and is associated with the index addr_hndl. The value of addr_hndl must be in the range 1 <= addr_hndl < LOC_ADDRTBL_SZ.

Return values

LAPI_SUCCESS    Indicates that the function call completed successfully.

LAPI_ERR_ADDR_HNDL_RANGE    Indicates that the value of addr_hndl is not in the
LAPI_Addr_get

range
1 <= addr_hndl < LOC_ADDRTBL_SZ.

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in terminated state).

LAPI_ERR_RET_PTR_NULL Indicates that the value of the addr pointer is Null (in C) or that the value of addr is LAPI_ADDR_NULL (in FORTRAN).

Location

/usr/lib/liblapi_r.a

C examples

To retrieve a header handler address that was previously registered using LAPI_Addr_set:

```c
lapi_handle_t hndl;    /* the LAPI handle */
void** addr;           /* the address to retrieve */
int    addr_hndl;      /* the index returned from LAPI_Addr_set */

...

addr_hndl = 1;
LAPI_Addr_get(hndl, &addr, addr_hndl);

/* addr now contains the address that was previously registered */
/* using LAPI_Addr_set */
```

Related information

Subroutines: LAPI_Addr_set, LAPI_Qenv
LAPI_Addr_set

Purpose

Registers the address of a function.

Library

Availability Library (liblapi_r.a)

C syntax

#include <lapi.h>

int LAPI_Addr_set(hndl, addr, addr_hndl)

lapi_handle_t hndl;

void *addr;

int addr_hndl;

FORTRAN syntax

include 'lapif.h'

lapi_addr_set(hndl, addr, addr_hndl, ierror)

integer hndl

integer (kind=LAPI_ADDR_TYPE) :: addr

integer addr_hndl

integer ierror

Parameters

Input

hndl Specifies the LAPI handle.

addr Specifies the address of the function handler that the user wants to register with LAPI.

addr_hndl Specifies a user function address that can be passed to LAPI calls in place of a header handler address. The value of this parameter must be in the range 1 <= addr_hndl < LOC_ADDRTBL_SZ.

Output

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: local address manipulation

Use this subroutine to register the address of a function (addr). LAPI maintains the function address in an internal table. The function address is indexed at location addr_hndl. In subsequent LAPI calls, addr_hndl can be used in place of addr. The value of addr_hndl must be in the range 1 <= addr_hndl < LOC_ADDRTBL_SZ.

For active message communication, you can use addr_hndl in place of the corresponding header handler address. LAPI only supports this indexed substitution for remote header handler addresses (but not other remote addresses, such as target counters or base data addresses). For these other types of addresses, the actual address value must be passed to the API call.
LAPI_Addr_set

This subroutine supports separate index spaces for multiple protocols. All protocols in a LAPI context get the same number of address indices and the number can be queried using `LAPI_Qenv(LOC_ADDRTBL_SZ)`.

Return values

**LAPI_SUCCESS**
Indicates that the function call completed successfully.

**LAPI_ERR_ADDR_HNDL_RANGE**
Indicates that the value of `addr_hndl` is not in the range `1 <= addr_hndl < LOC_ADDRTBL_SZ`.

**LAPI_ERR_HNDL_INVALID**
Indicates that the `hndl` passed in is not valid (not initialized or in terminated state).

Location

`/usr/lib/liblapi_r.a`

C examples

To register a header handler address:

```c
lapi_handle_t hndl; /* the LAPI handle */
void *addr; /* the remote header handler address */
int addr_hndl; /* the index to associate */

...

addr = my_func;
addr_hndl = 1;
LAPI_Addr_set(hndl, addr, addr_hndl);

/* addr_hndl can now be used in place of addr in LAPI_Amsend, */
/* LAPI_Amsendv, and LAPI_Xfer calls */
...
```

Related information

Subroutines: `LAPI_Addr_get`, `LAPI_Amsend`, `LAPI_Amsendv`, `LAPI_Qenv`, `LAPI_Xfer`
LAPI_Address

Purpose

Returns an unsigned long value for a specified user address.

Library

Availability Library (liblapi_r.a)

C syntax

```c
#include <lapi.h>

int LAPI_Address(my_addr, ret_addr)
void *my_addr;
ulong *ret_addr;
```

Note: This subroutine is meant to be used by FORTRAN programs. The C version of LAPI_Address is provided for compatibility purposes only.

FORTRAN syntax

```fortran
include 'lapif.h'

lapi_address(my_addr, ret_addr, ierror)
integer (kind=any_fortran_type) :: my_addr
integer (kind=LAPI_ADDR_TYPE) :: ret_addr
integer ierror
```

where:

any_fortran_type

Is any FORTRAN datatype. This type declaration has the same meaning as the type void * in C.

Parameters

Input

my_addr

Specifies the address to convert. The value of this parameter cannot be Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

Output

ret_addr

Returns the address that is stored in my_addr as an unsigned long for use in LAPI calls. The value of this parameter cannot be Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

derror

Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: local address manipulation

Use this subroutine in FORTRAN programs when you need to store specified addresses in an array. In FORTRAN, the concept of address (&) does not exist as it does in C. lapi_address provides FORTRAN programmers with this function.
Return values

LAPI_SUCCESS  Indicates that the function call completed successfully.

LAPI_ERR_ORG_ADDR_NULL  Indicates that the value of my_addr is Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_TGT_ADDR_NULL  Indicates that the value of ret_addr is Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

Location

/usr/lib/liblapi_r.a

FORTRAN examples

To retrieve the address of a variable:

```fortran
! Contains the address of the target counter
integer (kind=LAPI_ADDR_TYPE) :: cntr_addr

! Target counter
type (lapi_cntr_t) :: tgt_cntr

! Return code
integer :: ierror

call lapi_address(tgt_cntr, cntr_addr, ierror)

! cntr_addr now contains the address of tgt_cntr
```

Related information

Subroutines: LAPI_Address_init, LAPI_Address_init64
LAPI_Address_init

Purpose

Creates a remote address table.

Library

Availability Library (liblapi_r.a)

C syntax

```c
#include <lapi.h>

int LAPI_Address_init(hndl, my_addr, add_tab)
{
    lapi_handle_t hndl;
    void *my_addr;
    void *add_tab[];
}
```

FORTRAN syntax

```fortran
include 'lapif.h'

lapi_address_init(hndl, my_addr, add_tab, ierror)

integer hndl
integer (kind=LAPI_ADDR_TYPE) :: my_addr
integer (kind=LAPI_ADDR_TYPE) :: add_tab(*)
integer ierror
```

Parameters

Input

- **hndl**
  - Specifies the LAPI handle.

- **my_addr**
  - Specifies the entry supplied by each task. The value of this parameter can be Null (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

Output

- **add_tab**
  - Specifies the address table containing the addresses that are to be supplied by all tasks. **add_tab** is an array of pointers, the size of which is greater than or equal to **NUM_TASKS**. The value of this parameter cannot be Null (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

- **ierror**
  - Specifies a FORTRAN return code. This is always the last parameter.

Description

**Type of call:** collective communication (blocking)

**LAPI_Address_init** exchanges virtual addresses among tasks of a parallel application. Use this subroutine to create tables of such items as header handlers, target counters, and data buffer addresses.

**LAPI_Address_init** is a **collective call** over the LAPI handle **hndl**, which fills the table **add_tab** with the virtual address entries that each task supplies. Collective calls must be made in the same order at all participating tasks.
LAPI_Address_init

The addresses that are stored in the table *add_tab* are passed in using the *my_addr* parameter. Upon completion of this call, *add_tab[i]* contains the virtual address entry that was provided by task *i*. The array is opaque to the user.

Return values

- **LAPI_SUCCESS**: Indicates that the function call completed successfully.
- **LAPI_ERR_COLLECTIVE_PSS**: Indicates that a collective call was made while in persistent subsystem (PSS) mode.
- **LAPI_ERR_HNDL_INVALID**: Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).
- **LAPI_ERR_RET_PTR_NULL**: Indicates that the value of the *add_tab* pointer is Null (in C) or that the value of *add_tab* is **LAPI_ADDR_NULL** (in FORTRAN).

Location

```
/usr/lib/liblapi_r.a
```

C examples

To collectively transfer target counter addresses for use in a communication API call, in which all nodes are either 32-bit or 64-bit:

```c
lapi_handle_t hndl; // the LAPI handle
void *addr_tbl[NUM_TASKS]; // the table for all tasks' addresses
lapi_cntr_t tgt_cntr; // the target counter

...:

LAPI_Address_init(hndl, (void *)&tgt_cntr, addr_tbl);

/* for communication with task *t*, use addr_tbl[*t] */
/* as the address of the target counter */

...:
```

For a combination of 32-bit and 64-bit nodes, use **LAPI_Address_init64**.

Related information

Subroutines: **LAPI_Address**, **LAPI_Address_init64**
LAPI_Address_init64

Purpose

Creates a 64-bit remote address table.

Library

Availability Library (liblapi_r.a)

C syntax

```c
#include <lapi.h>

int LAPI_Address_init64(hndl, my_addr, add_tab)
    lapi_handle_t hndl;
    lapi_long_t my_addr;
    lapi_long_t *add_tab;
```

FORTRAN syntax

```fortran
include 'lapif.h'

lapi_address_init64(hndl, my_addr, add_tab, ierror)
    integer hndl!
    integer (kind=LAPI_ADDR_TYPE) :: my_addr
    integer (kind=LAPI_LONG_LONG_TYPE) :: add_tab(*)
    integer ierror
```

Parameters

Input

hndl Specifies the LAPI handle.
my_addr Specifies the address entry that is supplied by each task. The value of this parameter can be Null (in C) or LAPI_ADDR_NULL (in FORTRAN). To ensure 32-bit/64-bit interoperability, it is passed as a lapi_long_t type in C.

Output

add_tab Specifies the 64-bit address table that contains the 64-bit values supplied by all tasks. add_tab is an array of type lapi_long_t (in C) or LAPI_LONG_LONG_TYPE (in FORTRAN). The size of add_tab is greater than or equal to NUM_TASKS. The value of this parameter cannot be Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: collective communication (blocking)

LAPI_Address_init64 exchanges virtual addresses among a mixture of 32-bit and 64-bit tasks of a parallel application. Use this subroutine to create 64-bit tables of such items as header handlers, target counters, and data buffer addresses.
LAPI_Address_init64

LAPI_Address_init64 is a collective call over the LAPI handle hndl, which fills the
64-bit table add_tab with the virtual address entries that each task supplies.
Collective calls must be made in the same order at all participating tasks.

The addresses that are stored in the table add_tab are passed in using the
my_addr parameter. Upon completion of this call, add_tab[i] contains the virtual
address entry that was provided by task i. The array is opaque to the user.

Return values

LAPI_SUCCESS Indicates that the function call completed
successfully.

LAPI_ERR_COLLECTIVE_PSS Indicates that a collective call was made while in
persistent subsystem (PSS) mode.

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not
initialized or in terminated state).

LAPI_ERR_RET_PTR_NULL Indicates that the value of the add_tab pointer is
Null (in C) or that the value of add_tab is
LAPI_ADDR_NULL (in FORTRAN).

Location

/usr/lib/liblapi_r.a

C examples

To collectively transfer target counter addresses for use in a communication API call
with a mixed task environment (any combination of 32-bit and 64-bit):

```c
lapi_handle_t hndl; /* the LAPI handle */
lapi_long_t addr_tbl[NUM_TASKS]; /* the table for all tasks' addresses */
lapi_long_t tgt_cntr; /* the target counter */

LAPI_Address_init64(hndl, (lapi_long_t)&tgt_cntr, addr_tbl);

/* For communication with task t, use addr_tbl[t] as the address */
/* of the target counter. For mixed (32-bit and 64-bit) jobs, */
/* use the LAPI_Xfer subroutine for communication. */
```

Related information

Subroutines: LAPI_Address, LAPI_Address_init, LAPI_Xfer
**LAPI_Amsend**

**Purpose**

Transfers a user message to a remote task, obtaining the target address on the remote task from a user-specified header handler.

**Library**

Availability Library (liblapi_r.a)

**C syntax**

```c
#include <lapi.h>

typedef void (compl_hndlr_t) (hndl, user_info);

lapi_handle_t *hndl;  /* pointer to LAPI context passed in from LAPI_Amsend */
void *user_info;  /* buffer (user_info) pointer passed in */
/* from header handler (void *(hdr_hndlr_t)) */

typedef void *(hdr_hndlr_t)(hndl, uhdr, uhdr_len, msg_len, comp_h, user_info);

lapi_handle_t *hndl;  /* pointer to LAPI context passed in from LAPI_Amsend */
void *uhdr;  /* uhdr passed in from LAPI_Amsend */
uint *uhdr_len;  /* uhdr_len passed in from LAPI_Amsend */
ulong *msg_len;  /* msg_len passed in from LAPI_Amsend */
compl_hndlr_t **comp_h;  /* function address of completion handler */
/* (void (compl_hndlr_t)) that needs to be filled */
/* out by this header handler function */
void **user_info;  /* pointer to the parameter to be passed */
/* in to the completion handler */

int LAPI_Amsend(hndl, tgt, hdr_hdl, uhdr, uhdr_len, udata, udata_len,
                  tgt_cntr, org_cntr, cmpl_cntr);

lapi_handle_t hndl;
uint tgt;
void *hdr_hdl;
void *uhdr;
uint uhdr_len;
void *udata;
ulong udata_len;
lapi_cntr_t *tgt_cntr;
lapi_cntr_t *org_cntr;
lapi_cntr_t *cmpl_cntr;
```

**FORTRAN syntax**

```fortran
include 'lapif.h'

integer subroutine compl_h (hndl, user_info)
integer hndl
integer user_info

integer function hdr_hdl (hndl, uhdr, uhdr_len, msg_len, comp_h, user_info)
integer hndl
integer uhdr
integer uhdr_len
integer (kind=LAPI_LONG_TYPE) :: msg_len
external integer function comp_h
    type (LAPI_ADDR_TYPE) :: user_info

lapi_amsend(hndl, tgt, hdr_hdl, uhdr, uhdr_len, udata, udata_len,
```

168 IBM RSCT: LAPI Programming Guide
LAPI_Amsend

tgt_cntr, org_cntr, cmpl_cntr, ierror)
integer hndl
integer tgt
external integer function hdr_hdl
integer uhdr
integer uhdr_len
type (LAPI_ADDR_TYPE) :: udata
integer (kind=LAPI_LONG_TYPE) :: udata_len
integer (kind=LAPI_ADDR_TYPE) :: tgt_cntr
type (lapi_cntr_t) :: org_cntr
type (lapi_cntr_t) :: cmpl_cntr
integer ierror

Parameters

Input

hndl     Specifies the LAPI handle.
tgt      Specifies the task ID of the target task. The value of this parameter must be in the range \(0 \leq tgt < \text{NUM_TASKS}\).

hdr_hdl  Specifies the pointer to the remote header handler function to be invoked at the target. The value of this parameter can take an address handle that has already been registered using LAPI_Addr_set. The value of this parameter cannot be Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

uhdr     Specifies the pointer to the user header data. This data will be passed to the user header handler on the target. If \(\text{uhdr_len} = 0\), The value of this parameter can be Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

uhdr_len Specifies the length of the user’s header. The value of this parameter must be a multiple of the processor’s word size in the range \(0 \leq \text{uhdr_len} \leq \text{MAX_UHDR_SZ}\).

userdata Specifies the pointer to the user data. If \(\text{userdata_len} = 0\), The value of this parameter can be Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

userdata_len Specifies the length of the user data in bytes. The value of this parameter must be in the range \(0 \leq \text{userdata_len} \leq \text{value of LAPI constant LAPI_MAX_MSG_SZ}\).

Input/output
tgt_cntr Specifies the target counter address. The target counter is incremented after the completion handler (if specified) completes or after the completion of data transfer. If the value of this parameter is Null (in C) or LAPI_ADDR_NULL (in FORTRAN), the target counter is not updated.

org_cntr Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented after data is copied out of the origin address (in C) or the origin (in FORTRAN). If the value of this parameter is Null (in C) or LAPI_ADDR_NULL (in FORTRAN), the origin counter is not updated.

cmpl_cntr Specifies the counter at the origin that signifies completion of the completion handler. It is updated once the completion handler completes. If no completion handler is specified, the counter is incremented at the completion of message delivery. If the value of
**LAPI_Amsend**

this parameter is Null (in C) or **LAPI_ADDR_NULL** (in FORTRAN), the completion counter is not updated.

**Output**

*ierror* Specifies a FORTRAN return code. This is always the last parameter.

**Description**

**Type of call:** point-to-point communication (non-blocking)

Use this subroutine to transfer data to a target task, where it is desirable to run a handler on the target task before message delivery begins or after delivery completes. **LAPI_Amsend** allows the user to provide a header handler and optional completion handler. The header handler is used to specify the target buffer address for writing the data, eliminating the need to know the address on the origin task when the subroutine is called.

User data (*uhdr* and *udata*) are sent to the target task. Once these buffers are no longer needed on the origin task, the origin counter is incremented, which indicates the availability of origin buffers for modification. Using the **LAPI_Xfer** call with the **LAPI_AM_XFER** type provides the same type of transfer, with the option of using a send completion handler instead of the origin counter to specify buffer availability.

Upon arrival of the first data packet at the target, the user's header handler is invoked. Note that a header handler must be supplied by the user because it returns the base address of the buffer in which LAPI will write the data sent from the origin task (*udata*). See “Receive-side optimization for single-packet messages” on page 106 for more information.

The header handler also provides additional information to LAPI about the message delivery, such as the completion handler. **LAPI_Amsend** and similar calls (such as **LAPI_Amsendv** and corresponding **LAPI_Xfer** transfers) also allow the user to specify their own message header information, which is available to the header handler. The user may also specify a completion handler parameter from within the header handler. LAPI will pass the information to the completion handler at execution.

Note that the header handler is run inline by the thread running the LAPI dispatcher. For this reason, the header handler must be non-blocking because no other progress on messages will be made until it returns. It is also suggested that execution of the header handler be simple and quick. The completion handler, on the other hand, is normally enqueued for execution by a separate thread. It is possible to request that the completion handler be run inline. See “Inline handlers” on page 97 for more information.

If a completion handler was not specified (that is, set to **LAPI_ADDR_NULL** in FORTRAN or its pointer set to Null in C), the arrival of the final packet causes LAPI to increment the target counter on the remote task and send an internal message back to the origin task. The message causes the completion counter (if it is not Null in C or **LAPI_ADDR_NULL** in FORTRAN) to increment on the origin task.

If a completion handler was specified, the above steps take place after the completion handler returns. To guarantee that the completion handler has executed on the target, you must wait on the completion counter. See “Flow of active message operations” on page 73 for more information.
User details

As mentioned above, the user must supply the address of a header handler to be executed on the target upon arrival of the first data packet. The signature of the header handler is as follows:

```c
void *hdr_hndlr(lapi_handle_t *hndl, void *uhdr, uint *uhdr_len, ulong *msg_len,
                compl_hndlr_t **cmpl_hndlr, void **user_info);
```

The value returned by the header handler is interpreted by LAPI as an address for writing the user data (udata) that was passed to the LAPI_Amsend call. The uhdr and uhdr_len parameters are passed by LAPI into the header handler and contain the information passed by the user to the corresponding parameters of the LAPI_Amsend call.

Use of LAPI.Addr_set

Remote addresses are commonly exchanged by issuing a collective LAPI_Address_init call within a few steps of initializing LAPI. LAPI also provides the LAPI.Addr_set mechanism, whereby users can register one or more header handler addresses in a table, associating an index value with each address. This index can then be passed to LAPI_Amsend instead of an actual address. On the target side, LAPI will use the index to get the header handler address. Note that, if all tasks use the same index for their header handler, the initial collective communication can be avoided. Each task simply registers its own header handler address using the well-known index. Then, on any LAPI_Amsend calls, the reserved index can be passed to the header handler address parameter.

Role of the header handler

The user optionally returns the address of a completion handler function through the cmpl_hndlr parameter and a completion handler parameter through the user_info parameter. The address passed through the user_info parameter can refer to memory containing a datatype defined by the user and then cast to the appropriate type from within the completion handler if desired.

The signature for a user completion handler is as follows:

```c
typedef void (compl_hndlr_t)(lapi_handle_t *hndl, void *completion_param);
```

The argument returned by reference through the user_info member of the user’s header handler will be passed to the completion_param argument of the user’s completion handler. See the C Examples for an example of setting the completion handler and parameter in the header handler.

As mentioned above, the value returned by the header handler must be an address for writing the user data sent from the origin task. There is one exception to this rule. In the case of a single-packet message, LAPI passes the address of the packet in the receive FIFO, allowing the entire message to be consumed within the header handler. In this case, the header handler should return NULL (in C) or LAPI_ADDR_NULL (in FORTRAN) so that LAPI does not copy the message to a target buffer. See “Receive-side optimization for single-packet messages” on page 106 for more information.
Passing additional information through lapi_return_info_t

LAPI allows additional information to be passed to and returned from the header handler by passing a pointer to lapi_return_info_t through the msg_len argument. On return from a header handler that is invoked by a call to LAPI_Amsend, the ret_flags member of lapi_return_info_t can contain one of these values:

- LAPI_NORMAL (the default),
- LAPI_SEND_REPLY (to run the completion handler inline),
- LAPI_LOCAL_STATE (no reply is sent).

The dgsp_handle member of lapi_return_info_t should not be used in conjunction with LAPI_Amsend.

For a complete description of the lapi_return_info_t type, see "The enhanced header handler interface" on page 95.

Inline execution of completion handlers

Under normal operation, LAPI uses a separate thread for executing user completion handlers. After the final packet arrives, completion handler pointers are placed in a queue to be handled by this thread. For performance reasons, the user may request that a given completion handler be run inline instead of being placed on this queue behind other completion handlers. This mechanism gives users a greater degree of control in prioritizing completion handler execution for performance-critical messages.

LAPI places no restrictions on completion handlers that are run "normally" (that is, by the completion handler thread). Inline completion handlers should be short and should not block, because no progress can be made while the main thread is executing the handler. The user must use caution with inline completion handlers so that LAPI's internal queues do not fill up while waiting for the handler to complete. I/O operations must not be performed with an inline completion handler.

Return values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_DATA_LEN Indicates that the value of udata_len is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_HDR_HNDLR_NULL Indicates that the value of the hdr_hdl passed in is Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in terminated state).

LAPI_ERR_ORG_ADDR_NULL Indicates that the value of the udata parameter passed in is Null (in C) or LAPI_ADDR_NULL (in FORTRAN), but the value of udata_len is greater than 0.

LAPI_ERR_TGT Indicates that the tgt passed in is outside the range of tasks defined in the job.

LAPI_ERR_TGT_PURGED Indicates that the subroutine returned early because LAPI_Purge_totask() was called.

LAPI_ERR_UHDR_LEN Indicates that the uhdr_len value passed in is greater than MAX_UHDR_SZ or is not a multiple of the processor's word size.
**LAPI_Amsend**

**LAPI_ERR_UHDR_NULL** Indicates that the `uhdr` passed in is Null (in C) or `LAPI_ADDR_NULL` (in FORTRAN), but `uhdr_len` is not 0.

**Location**

`/usr/lib/liblapi_r.a`

**C examples**

To send an active message and then wait on the completion counter:

```c
/* header handler routine to execute on target task */
void *hdr_hndlr(lapi_handle_t *hndl, void *uhdr, uint *uhdr_len,
    ulong *msg_len, compl_hndlr_t **cmpl_hndlr,
    void **user_info)
{
    /* set completion handler pointer and other information */
    /* return base address for LAPI to begin its data copy */
}
```

```c
lapi_handle_t hndl; /* the LAPI handle */
int task_id; /* the LAPI task ID */
int num_tasks; /* the total number of tasks */
void *hdr_hndlr_list[NUM_TASKS]; /* the table of remote header handlers */
int buddy; /* the communication partner */
lapi_cntr_t cmpl_cntr; /* the completion counter */
int data_buffer[DATA_LEN]; /* the data to transfer */
```

```c
    /* retrieve header handler addresses */
    LAPI_Address_init(hndl, (void *)&hdr_hndlr, hdr_hndlr_list);

    /* up to this point, all instructions have executed on all
    ** tasks. we now begin differentiating tasks. */
    if ( sender ) { /* origin task */
        /* initialize data buffer, cmpl_cntr, etc. */
        .
        .
        /* synchronize before starting data transfer */
        LAPI_Gfence(hndl);

        LAPI_Amsend(hndl, buddy, (void *)&hdr_hndlr_list[buddy], Null,
            0,&(data_buffer[0]),DATA_LEN*sizeof(int),
            Null, Null, cmpl_cntr);

        /* Wait on completion counter before continuing. Completion */
        /* counter will update when message completes at target. */
    } else { /* receiver */
        .
        .
        /* to match the origin's synchronization before data transfer */
        LAPI_Gfence(hndl);
    }
```
LAPI_Amsend

...}

For a complete program listing, see “Using LAPI_Amsend: a complete LAPI program” on page 74. Sample code illustrating the LAPI_Amsend call can be found in the LAPI sample files. See Chapter 22, “Sample LAPI programs,” on page 291 for more information.

Related information

Subroutines: LAPI_Addr_get, LAPI_Addr_set, LAPI_Getcntr, LAPI_Msgpoll, LAPI_Qenv, LAPI_Setcntr, LAPI_Waitcntr, LAPI_Xfer
LAPI_Amsendv

Purpose

Transfers a user vector to a remote task, obtaining the target address on the remote task from a user-specified header handler.

Library

Availability Library (liblapi_r.a)

C syntax

```c
#include <lapi.h>

typedef void (compl_hndlr_t) (hndl, user_info);

lapi_handle_t *hndl; /* the LAPI handle passed in from LAPI_Amsendv */
void *user_info; /* the buffer (user_info) pointer passed in */
/* from vhdr_hndlr (void *(vhdr_hndlr_t)) */

typedef lapi_vec_t *(vhdr_hndlr_t) (hndl, uhdr, uhdr_len, len_vec, comp_h, uinfo);

lapi_handle_t *hndl; /* pointer to the LAPI handle passed in from LAPI_Amsendv */
void *uhdr; /* uhdr passed in from LAPI_Amsendv */
uint *uhdr_len; /* uhdr_len passed in from LAPI_Amsendv */
ulong *len_vec[ ]; /* vector of lengths passed in LAPI_Amsendv */
compl_hndlr_t **comp_h; /* function address of completion handler */
/* (void (compl_hndlr_t)) that needs to be */
/* filled out by this header handler function */
void **user_info; /* pointer to the parameter to be passed */
/* in to the completion handler */

int LAPI_Amsendv(hndl, tgt, hdr_hdl, uhdr, uhdr_len, org_vec,
                 tgt_cntr, org_cntr, cmpl_cntr);

lapi_handle_t  hndl;
uint  tgt;
void  *hdr_hdl;
void  *uhdr;
uint  uhdr_len;
lapi_vec_t  *org_vec;
lapi_cntr_t  *tgt_cntr;
lapi_cntr_t  *org_cntr;
lapi_cntr_t  *cmpl_cntr;
```

FORTRAN syntax

```fortran
include 'lapif.h'

integer subroutine compl_h (hndl, user_info)
integer hndl
integer user_info(*)

integer function vhdr_hdl (hndl, uhdr, uhdr_len, len_vec, comp_h, user_info)
integer hndl
integer uhdr
integer uhdr_len
integer (kind=LAPI_LONG_TYPE) :: len_vec
external integer function comp_h
type (LAPI_ADDR_TYPE) :: user_info

lapi_amsendv(hndl, tgt, hdr_hdl, uhdr, uhdr_len, org_vec,
             tgt_cntr, org_cntr, cmpl_cntr, ierror)
integer hndl
integer tgt
```
LAPI_Amsendv

```fortran
external integer function hdr_hdl
  integer uhdr
  integer uhdr_len
  type (lapi_vec_t) :: org_vec
  integer (kind=LAPI_ADDR_TYPE) :: tgt_cntr
  type (lapi_cntr_t) :: org_cntr
  type (lapi_cntr_t) :: cmpl_cntr
  integer ierror
```

**Parameters**

- **hndl**
  Specifies the LAPI handle.

- **tgt**
  Specifies the task ID of the target task. The value of this parameter must be in the range 0 <= tgt < NUM_TASKS.

- **hdr_hdl**
  Points to the remote header handler function to be invoked at the target. The value of this parameter can take an address handle that had been previously registered using the LAPI_Addr_set/LAPI_Addr_get mechanism. The value of this parameter cannot be Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

- **uhdr**
  Specifies the pointer to the local header (parameter list) that is passed to the handler function. If uhdr_len is 0, The value of this parameter can be Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

- **uhdr_len**
  Specifies the length of the user’s header. The value of this parameter must be a multiple of the processor’s word size in the range 0 <= uhdr_len <= MAX_UHDR_SZ.

- **org_vec**
  Points to the origin vector.

**Input/output**

- **tgt_cntr**
  Specifies the target counter address. The target counter is incremented after the completion handler (if specified) completes or after the completion of data transfer. If the value of this parameter is Null (in C) or LAPI_ADDR_NULL, the target counter is not updated.

- **org_cntr**
  Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented after data is copied out of the origin address (in C) or the origin (in FORTRAN). If the value of this parameter is Null (in C) or LAPI_ADDR_NULL (in FORTRAN), the origin counter is not updated.

- **cmpl_cntr**
  Specifies the counter at the origin that signifies completion of the completion handler. It is updated once the completion handler completes. If no completion handler is specified, the counter is incremented at the completion of message delivery. If the value of this parameter is Null (in C) or LAPI_ADDR_NULL (in FORTRAN), the completion counter is not updated.

**Output**

- **ierror**
  Specifies a FORTRAN return code. This is always the last parameter.

**Description**

*Type of call:* point-to-point communication (non-blocking)
**LAPI_Amsendv**

LAPI_Amsendv is the vector-based version of the LAPI_Amsend call. You can use it to specify multi-dimensional and non-contiguous descriptions of the data to transfer. Whereas regular LAPI calls allow the specification of a single data buffer address and length, the vector versions allow the specification of a vector of address and length combinations. Additional information is allowed in the data description on the origin task and the target task.

Use this subroutine to transfer a vector of data to a target task, when you want a handler to run on the target task before message delivery begins or after message delivery completes.

To use LAPI_Amsendv, you must provide a header handler, which returns the address of the target vector description that LAPI uses to write the data that is described by the origin vector. The header handler is used to specify the address of the vector description for writing the data, which eliminates the need to know the description on the origin task when the subroutine is called. The header handler is called upon arrival of the first data packet at the target.

Optionally, you can also provide a completion handler. The header handler provides additional information to LAPI about the message delivery, such as the completion handler. You can also specify a completion handler parameter from within the header handler. LAPI passes the information to the completion handler at execution.

With the exception of the address that is returned by the completion handler, the use of counters, header handlers, and completion handlers in LAPI_Amsendv is identical to that of LAPI_Amsend. In both cases, the user header handler returns information that LAPI uses for writing at the target. See LAPI_Amsend for more information.

This is a non-blocking call. The calling task cannot change the uhdr (origin header) and org_vec data until completion at the origin is signaled by the org_cntr being incremented. The calling task cannot assume that the org_vec structure can be changed before the origin counter is incremented. The structure (of type lapi_vec_t) that is returned by the header handler cannot be modified before the target counter has been incremented. Also, if a completion handler is specified, it may execute asynchronously, and can only be assumed to have completed after the target counter increments (on the target) or the completion counter increments (at the origin).

The length of the user-specified header (uhdr_len) is constrained by the implementation-specified maximum value MAX_UHDR_SZ. uhdr_len must be a multiple of the processor's word size. To get the best bandwidth, uhdr_len should be as small as possible.

If the following requirement is not met, an error condition occurs:

- If a strided vector is being transferred, the size of each block must not be greater than the stride size in bytes.

LAPI does not check for any overlapping regions among vectors either at the origin or the target. If the overlapping regions exist on the target side, the contents of the target buffer are undefined after the operation.

**Return values**

LAPI_SUCCESS Indicates that the function call completed successfully.
LAPI_ERR_HDR_HNDLR_NULL
  Indicates that the hdr_hdl passed in is Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_HNDL_INVALID
  Indicates that the hndl passed in is not valid (not initialized or in terminated state).

LAPI_ERR_ORG_EXTENT
  Indicates that the org_vec's extent (stride * num_vecs) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_ORG_STRIDE
  Indicates that the org_vec stride is less than block.

LAPI_ERR_ORG_VEC_ADDR
  Indicates that the org_vec->info[i] is Null (in C) or LAPI_ADDR_NULL (in FORTRAN), but its length (org_vec->len[i]) is not 0.

LAPI_ERR_ORG_VEC_LEN
  Indicates that the sum of org_vec->len is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_ORG_VEC_NULL
  Indicates that org_vec is Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_ORG_VEC_TYPE
  Indicates that the org_vec->vec_type is not valid.

LAPI_ERR_STRIDE_ORG_VEC_ADDR_NULL
  Indicates that the strided vector address org_vec->info[0] is Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_TGT
  Indicates that the tgt passed in is outside the range of tasks defined in the job.

LAPI_ERR_TGT_PURGED
  Indicates that the subroutine returned early because LAPI_Purge_totask() was called.

LAPI_ERR_UHDR_LEN
  Indicates that the uhdr_len value passed in is greater than MAX_UHDR_SZ or is not a multiple of the processor's word size.

LAPI_ERR_UHDR_NULL
  Indicates that the uhdr passed in is Null (in C) or LAPI_ADDR_NULL (in FORTRAN), but uhdr_len is not 0.

Location
/usr/lib/liblapi_r.a

C examples

1. To send a LAPI_GEN_IOVECTOR using active messages:

   /* header handler routine to execute on target task */
   lapi_vec_t *hdr_hndlr(lapi_handle_t *handle, void *uhdr, uint *uhdr_len,
     ulong *len_vec[], compl_hndlr_t **completion_handler,
     void ***user_info)
   {
     /* set completion handler pointer and other info */

     /* set up the vector to return to LAPI */
     /* for a LAPI_GEN_IOVECTOR: num_vecs, vec_type, and len must all have */
     /* the same values as the origin vector. The info array should */
/* contain the buffer addresses for LAPI to write the data */
vec->num_vecs = NUM_VECS;
vec->vec_type = LAPI_GEN_IDVECTOR;
vec->len = (unsigned long *)malloc(NUM_VECS*sizeof(unsigned long));
vec->info = (void **) malloc(NUM_VECS*sizeof(void *));
for (i=0; i < NUM_VECS; i++) {
    vec->info[i] = (void *) &data_buffer[i];
    vec->len[i] = (unsigned long)(sizeof(int));
}
return vec;

This example could also illustrate the LAPI_GEN_GENERIC type, with these modifications:

- Both vectors need a vec_type of LAPI_GEN_GENERIC.
- The size of the target side must be greater than or equal to the size of the origin side. There are no other restrictions on the symmetry of number of vectors and lengths between the origin side and the target side.

2. To send a LAPI_STRIDED_VECTOR using active messages:

/* header handler routine to execute on target task */
lapi_vec_t *hdr_hndlr(lapi_handle_t *handle, void *uhdr, uint *uhdr_len,
                     ulong *len_vec[], compl_hndlr_t **completion_handler,
                     void **user_info)
{
    int block_size;    /* block size */
    int data_size;     /* stride */
    ...
    vec->num_vecs = NUM_VECS;    /* NUM_VECS = number of vectors to transfer */
For complete examples, see the sample programs shipped with LAPI.

Related information
[Using vectors" on page 47] for information about vector data transfer

Subroutines: LAPI_Addr_get, LAPI_Addr_set, LAPI_Address_init,
LAPI_Amsend, LAPI_Getcnt, LAPI_Getv, LAPI_Putv, LAPI_Qenv,
LAPI_Waitcnt, LAPI_Xfer
LAPI_Fence

Purpose
Enforces order on LAPI calls.

Library
Availability Library (liblapi_r.a)

C syntax
#include <lapi.h>

int LAPI_Fence(hndl)
    lapi_handle_t hndl;

FORTRAN syntax
include 'lapif.h'

lapi_fence(hndl, ierror)
    integer hndl
    integer ierror

Parameters
Input
hndl
    Specifies the LAPI handle.

Output
ierror
    Specifies a FORTRAN return code. This is always the last parameter.

Description
Type of call: Local data synchronization (blocking) (may require progress on the remote task)

Use this subroutine to enforce order on LAPI calls. If a task calls LAPI_Fence, all the LAPI operations that were initiated by that task, before the fence using the LAPI context hndl, are guaranteed to complete at the target tasks. This occurs before any of its communication operations using hndl, initiated after the LAPI_Fence, start transmission of data. This is a data fence which means that the data movement is complete. This is not an operation fence which would need to include active message completion handlers completing on the target.

LAPI_Fence may require internal protocol processing on the remote side to complete the fence request.

Return values
LAPI_SUCCESS
    Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID
    Indicates that the hndl passed in is not valid (not initialized or in terminated state).
LAPI_Fence

Location

/usr/lib/liblapi_r.a

C examples

To establish a data barrier in a single task:

```c
lapi_handle_t hndl; /* the LAPI handle */

: /* API communication call 1 */
: /* API communication call 2 */

: /* API communication call n */
LAPI_Fence(hndl);

/* all data movement from above communication calls has completed by this point */
/* any completion handlers from active message calls could still be running. */
```

Related information

Subroutines: LAPI_Amsend, LAPI_Gfence
LAPI_Get

Purpose

Copies data from a remote task to a local task.

Library

Availability Library (liblapi_r.a)

C syntax

```c
#include <lapi.h>
int LAPI_Get(hndl, tgt, len, tgt_addr, org_addr, tgt_cntr, org_cntr)
lapi_handle_t hndl;
uint tgt;
ulong len;
void *tgt_addr;
void *org_addr;
lapi_cntr_t *tgt_cntr;
lapi_cntr_t *org_cntr;
```

FORTRAN syntax

```fortran
include 'lapif.h'
lapi_get(hndl, tgt, len, tgt_addr, org_addr, tgt_cntr, org_cntr, ierr)
integer hndl
integer tgt
integer (kind=LAPI_LONG_TYPE) :: len
integer (kind=LAPI_ADDR_TYPE) :: tgt_addr
integer (kind=LAPI_ADDR_TYPE) :: org_addr
integer (kind=LAPI_ADDR_TYPE) :: tgt_cntr
type (lapi_cntr_t) :: org_cntr
integer ierr
```

Parameters

Input

- **hndl**: Specifies the LAPI handle.
- **tgt**: Specifies the task ID of the target task that is the source of the data. The value of this parameter must be in the range $0 \leq tgt < \text{NUM_TASKS}$. 
- **len**: Specifies the number of bytes of data to be copied. This parameter must be in the range $0 \leq len \leq \text{value of LAPI constant LAPI_MAX_MSG_SZ}$. 
- **tgt_addr**: Specifies the target buffer address of the data source. If len is 0, the value of this parameter can be Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

Input/output

- **tgt_cntr**: Specifies the target counter address. The target counter is incremented once the data buffer on the target can be modified. If the value of this parameter is Null (in C) or LAPI_ADDR_NULL (in FORTRAN), the target counter is not updated.
- **org_cntr**: Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented after data arrives at
LAPI_Get

the origin. If the value of this parameter is Null (in C) or LAPI_ADDR_NULL (in FORTRAN), the origin counter is not updated.

Output

org_addr Specifies the local buffer address into which the received data is copied. If len is 0, The value of this parameter can be Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: point-to-point communication (non-blocking)

Use this subroutine to transfer data from a remote (target) task to a local (origin) task. Note that in this case the origin task is actually the receiver of the data. This difference in transfer type makes the counter behavior slightly different than in the normal case of origin sending to target.

The origin buffer will still increment on the origin task upon availability of the origin buffer. But in this case, the origin buffer becomes available once the transfer of data is complete. Similarly, the target counter will increment once the target buffer is available. Target buffer availability in this case refers to LAPI no longer needing to access the data in the buffer.

This is a non-blocking call. The caller cannot assume that data transfer has completed upon the return of the function. Instead, counters should be used to ensure correct buffer addresses as defined above.

Note that a zero-byte message does not transfer data, but it does have the same semantic with respect to counters as that of any other message.

Return values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_DATA_LEN Indicates that the value of udata_len is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in terminated state).

LAPI_ERR_ORG_ADDR_NULL Indicates that the org_addr passed in is Null (in C) or LAPI_ADDR_NULL (in FORTRAN), but len is greater than 0.

LAPI_ERR_TGT Indicates that the tgt passed in is outside the range of tasks defined in the job.

LAPI_ERR_TGT_ADDR_NULL Indicates that the tgt_addr passed in is Null (in C) or LAPI_ADDR_NULL (in FORTRAN), but len is greater than 0.

LAPI_ERR_TGT_PURGED Indicates that the subroutine returned early because LAPI_Purge_totask() was called.
Location

/usr/lib/liblapi_r.a

C examples

{
    /* initialize the table buffer for the data addresses */
    /* get remote data buffer addresses */
    LAPI_Address_init(hndl,(void *)data_buffer,data_buffer_list);
    .
    .
    LAPI_Get(hndl, tgt, (ulong) data_len, (void *) (data_buffer_list[tgt]),
             (void *) data_buffer, tgt_cntr, org_cntr);
    /* retrieve data_len bytes from address data_buffer_list[tgt] on task tgt. */
    /* write the data starting at address data_buffer. tgt_cntr and org_cntr */
    /* can be Null. */
}

Related information
Subroutines: LAPI_Address_init, LAPI_Getcntr, LAPI_Put, LAPI_Qenv, LAPI_Waitcntr, LAPI_Xfer
LAPI_Getcntr

Purpose

Gets the integer value of a specified LAPI counter.

Library

Availability Library (liblapi_r.a)

C syntax

```c
#include <lapi.h>

int LAPI_Getcntr(hndl, cntr, val)
    lapi_handle_t *hndl;
    lapi_cntr_t *cntr;
    int *val;
```

FORTRAN syntax

```fortran
include 'lapif.h'

lapi_getcntr(hndl, cntr, val, ierror)
    integer hndl
    type (lapi_cntr_t) :: cntr
    integer val
    integer ierror
```

Parameters

**Input**

- **hndl**: Specifies the LAPI handle.
- **cntr**: Specifies the address of the counter. The value of this parameter cannot be Null (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

**Output**

- **val**: Returns the integer value of the counter `cntr`. The value of this parameter cannot be Null (in C) or `LAPI_ADDR_NULL` (in FORTRAN).
- **ierror**: Specifies a FORTRAN return code. This is always the last parameter.

Description

**Type of call**: Local counter manipulation

This subroutine gets the integer value of `cntr`. It is used to check progress on `hndl`.

Return values

- **LAPI_SUCCESS**: Indicates that the function call completed successfully.
- **LAPI_ERR_CNTR_NULL**: Indicates that the `cntr` pointer is Null (in C) or that the value of `cntr` is `LAPI_ADDR_NULL` (in FORTRAN).
**LAPI_Getcntr**

**LAPI_ERR_HNDL_INVALID**  Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

**LAPI_ERR_RET_PTR_NULL**  Indicates that the value of the *val* pointer is Null (in C) or that the value of *val* is **LAPI_ADDR_NULL** (in FORTRAN).

**Location**

/usr/lib/liblapi_r.a

**C examples**

```c
{
    lapi_cntr_t cntr;
    int val;

    /* cntr is initialized */
    /* processing/communication takes place */
    LAPI_Getcntr(hndl, &cntr, &val)
    /* val now contains the current value of cntr */
}
```

**Related information**

LAPI_Getv

Purpose
Copies vectors of data from a remote task to a local task.

Library
Availability Library (liblapi_r.a)

C syntax
#include <lapi.h>

int LAPI_Getv(
lapi_handle_t hndl;
uint tgt;
lapi_vec_t *tgt_vec;
lapi_vec_t *org_vec;
lapi_cntr_t *tgt_cntr;
lapi_cntr_t *org_cntr;

typedef struct {
    lapi_vectype_t vec_type; /* operation code */
    uint num_vecs; /* number of vectors */
    void **info; /* vector of information */
    ulong **len; /* vector of lengths */
} lapi_vec_t;

FORTRAN syntax
include 'lapif.h'

lapi_getv(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr, ierror)

The 32-bit version of the lapi_vec_t type is defined as:
type lapi_vec_t
    sequence
    integer(4) :: vec_type
    integer(4) :: num_vecs
    integer(4) :: info
    integer(4) :: len
end type lapi_vec_t

The 64-bit version of the lapi_vec_t type is defined as:
type lapi_vec_t
    sequence
    integer(8) :: vec_type
    integer(8) :: num_vecs
    integer(8) :: info
    integer(8) :: len
end type lapi_vec_t
LAPI_Getv

Parameters

Input

hndl Specifies the LAPI handle.

tgt Specifies the task ID of the target task. The value of this parameter must be in the range \(0 \leq tgt < \text{NUM\_TASKS}\).

tgt_vec Points to the target vector description.

org_vec Points to the origin vector description.

Input/output

tgt_cntr Specifies the target counter address. The target counter is incremented once the data buffer on the target can be modified. If the value of this parameter is Null (in C) or \text{LAPI\_ADDR\_NULL} (in FORTRAN), the target counter is not updated.

org_cntr Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented after data arrives at the origin. If the value of this parameter is Null (in C) or \text{LAPI\_ADDR\_NULL} (in FORTRAN), the origin counter is not updated.

Output

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: point-to-point communication (non-blocking)

This subroutine is the vector version of the \text{LAPI\_Get} call. Use \text{LAPI\_Getv} to transfer vectors of data from the target task to the origin task. Both the origin and target vector descriptions are located in the address space of the origin task. But, the values specified in the \text{info} array of the target vector must be addresses in the address space of the target task.

The calling program cannot assume that the origin buffer can be changed or that the contents of the origin buffers on the origin task are ready for use upon function return. After the origin counter (\text{org\_cntr}) is incremented, the origin buffers can be modified by the origin task. After the target counter (\text{tgt\_cntr}) is incremented, the target buffers can be modified by the target task. If you provide a completion counter (\text{cmpl\_cntr}), it is incremented at the origin after the target counter (\text{tgt\_cntr}) has been incremented at the target. If the values of any of the counters or counter addresses are Null (in C) or \text{LAPI\_ADDR\_NULL} (in FORTRAN), the data transfer occurs, but the corresponding counter increments do not occur.

If any of the following requirements are not met, an error condition occurs:

- The vector types \text{org\_vec->vec\_type} and \text{tgt\_vec->vec\_type} must be the same.
- If a strided vector is being transferred, the size of each block must not be greater than the stride size in bytes.
- The length of any vector that is pointed to by \text{tgt\_vec} must be equal to the length of the corresponding vector that is pointed to by \text{org\_vec}.
LAPI does not check for any overlapping regions among vectors either at the origin or the target. If the overlapping regions exist on the origin side, the contents of the origin buffer are undefined after the operation.

See LAPI_Amsendv for details about communication using different LAPI vector types. (LAPI_Getv does not support the LAPI_GEN_GENERIC type.)

Return values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in terminated state).

LAPI_ERR_ORG_EXTENT Indicates that the org_vec's extent (stride * num_vecs) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_ORG_STRIDE Indicates that the org_vec stride is less than block size.

LAPI_ERR_ORG_VEC_ADDR Indicates that some org_vec->info[i] is Null (in C) or LAPI_ADDR_NULL (in FORTRAN), but the corresponding length (org_vec->len[i]) is not 0.

LAPI_ERR_ORG_VEC_LEN Indicates that the total sum of all org_vec->len[i] (where [i] is in the range 0 <= i <= org_vec->num_vecs) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_ORG_VEC_NULL Indicates that the org_vec is Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_ORG_VEC_TYPE Indicates that the org_vec->vec_type is not valid.

LAPI_ERR_STRIDE_ORG_VEC_ADDR_Null Indicates that the strided vector base address org_vec->info[0] is Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_STRIDE_TGT_VEC_ADDR_Null Indicates that the strided vector address tgt_vec->info[0] is Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_TGT Indicates that the tgt passed in is outside the range of tasks defined in the job.

LAPI_ERR_TGT_EXTENT Indicates that tgt_vec’s extent (stride * num_vecs) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_TGT_PURGED Indicates that the subroutine returned early because LAPI_Purge_totask() was called.

LAPI_ERR_TGT_STRIDE Indicates that the tgt_vec’s stride is less than its block size.

LAPI_ERR_TGT_VEC_ADDR Indicates that the tgt_vec->info[i] is Null (in C) or LAPI_ADDR_NULL (in FORTRAN), but its length (tgt_vec->len[i]) is not 0.
**LAPI_Getv**

**LAPI_ERR_TGT_VEC_LEN** Indicates that the sum of `tgt_vec->len` is greater than the value of LAPI constant `LAPI_MAX_MSG_SZ`.

**LAPI_ERR_TGT_VEC_NULL** Indicates that `tgt_vec` is Null (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

**LAPI_ERR_TGT_VEC_TYPE** Indicates that the `tgt_vec->vec_type` is not valid.

**LAPI_ERR_VEC_LEN_DIFF** Indicates that `org_vec` and `tgt_vec` have different lengths (`len[i]`).

**LAPI_ERR_VEC_NUM_DIFF** Indicates that `org_vec` and `tgt_vec` have different `num_vecs`.

**LAPI_ERR_VEC_TYPE_DIFF** Indicates that `org_vec` and `tgt_vec` have different vector types (`vec_type`).

**Location**

`/usr/lib/liblapi_r.a`

**C examples**

To get a `LAPI_GEN_IOVECTOR`:

```c
{
    /* retrieve a remote data buffer address for data to transfer, */
    /* such as through LAPI_Address_init */

    /* task that calls LAPI_Getv sets up both org_vec and tgt_vec */
    org_vec->num_vecs = NUM_VECS;
    org_vec->vec_type = LAPI_GEN_IOVECTOR;
    org_vec->len = (unsigned long *) malloc(NUM_VECS*sizeof(unsigned long));
    org_vec->info = (void **) malloc(NUM_VECS*sizeof(void *));

    /* each org_vec->info[i] gets a base address on the origin task */
    /* each org_vec->len[i] gets the number of bytes to write to */
    /* org_vec->info[i] */

    tgt_vec->num_vecs = NUM_VECS;
    tgt_vec->vec_type = LAPI_GEN_IOVECTOR;
    tgt_vec->len = (unsigned long *) malloc(NUM_VECS*sizeof(unsigned long));
    tgt_vec->info = (void **) malloc(NUM_VECS*sizeof(void *));

    /* each tgt_vec->info[i] gets a base address on the target task */
    /* each tgt_vec->len[i] gets the number of bytes to transfer */
    /* from vec->info[i] */
    /* For LAPI_GEN_IOVECTOR, num_vecs, vec_type, and len must be */
    /* the same */

    LAPI_Getv(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr);
    /* tgt_cntr and org_cntr can both be Null */

    /* data will be retrieved as follows: */
    /* org_vec->len[0] bytes will be retrieved from */
    /* tgt_vec->info[0] and written to org_vec->info[0] */
    /* org_vec->len[1] bytes will be retrieved from */
    /* tgt_vec->info[1] and written to org_vec->info[1] */
    .
    .
}
```
LAPI_Getv

```c
/* org_vec->len[NUM_VECS-1] bytes will be retrieved */
/* from tgt_vec->info[NUM_VECS-1] and written to */
/* org_vec->info[NUM_VECS-1] */
```

For examples of other vector types, see LAPI_Amsendv.

**Related information**

Subroutines: LAPI_Amsendv, LAPI_Getcntr, LAPI_Putv, LAPI_Qenv, LAPI_Waitcntr
LAPI_Gfence

Purpose
Enforces order on LAPI calls across all tasks and provides barrier synchronization among them.

Library
Availability Library (liblapi_r.a)

C syntax
```c
#include <lapi.h>

int LAPI_Gfence(hndl)
    lapi_handle_t hndl;
```

FORTRAN syntax
```fortran
include 'lapif.h'

lapi_gfence(hndl, ierror)
    integer hndl
    integer ierror
```

Parameters

Input
- **hndl** Specifies the LAPI handle.

Output
- **ierror** Specifies a FORTRAN return code. This is always the last parameter.

Description

*Type of call:* collective data synchronization (blocking)

Use this subroutine to enforce global order on LAPI calls. This is a *collective call*. Collective calls must be made in the same order at all participating tasks.

On completion of this call, it is assumed that all LAPI communication associated with *hndl* from all tasks has quiesced. Although *hndl* is local, it represents a set of tasks that were associated with it at **LAPI_Init**, all of which must participate in this operation for it to complete. This is a data fence, which means that the data movement is complete. This is not an operation fence, which would need to include active message completion handlers completing on the target.

Return values

- **LAPI_SUCCESS** Indicates that the function call completed successfully.
- **LAPI_ERR_HNDL_INVALID** Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).
LAPI_Gfence

Location

/usr/lib/liblapi_r.a

Related information

Subroutines: LAPI_Fence
LAPI_Group_create

Purpose

Creates a multicast group and returns the group handle that is associated with the newly-created group.

Library

Availability Library (liblapi_r.a)

C syntax

```c
#include <lapi.h>

int LAPI_Group_create(hndl, tnum, *tlist, *group)

lapi_handle_t hndl;
uint tnum;
uint *tlist;
lapi_group_t *group;
```

FORTRAN syntax

```fortran
include 'lapif.h'

lapi_group_create(hndl, tnum, *tlist, *group, ierr)

integer hndl
integer tnum
integer tlist

type (lapi_group_t) :: group
integer ierr
```

Parameters

Input

- `hndl` Specifies the LAPI handle.
- `tnum` Specifies the number of tasks in this group.
- `tlist` Specifies the task ID of each member in this group (array of integers).

Output

- `group` Returns the group handle of the newly-created group.
- `ierror` Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: collective operation (blocking)

This subroutine returns the group handle associated with a newly-created multicast group. Only the members of the group can participate in the `LAPI_Group_create` operation.

Return values

- `LAPI_SUCCESS` Indicates that the function call completed successfully.
LAPI_Group_create

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in terminated state).
LAPI_ERR_NULL_GRP Indicates that the group passed in is Null.
LAPI_ERR_NULL_LIST Indicates that the task list passed in is Null.
LAPI_ERR_TASK_NUM Indicates that the number of tasks passed in is not valid (less than 0 or greater than or equal to num_tasks)

Location

/usr/lib/liblapi_r.a

C examples

To create a multicast group:

lapi_handle_t hndl; /* the LAPI handle */
lapi_group_t group; /* the group handle */
int task_num; /* number of tasks in the newly-created group */
int task_list[NUM_TASKS]; /* array to store task IDs */

task_num = 3;
task_list[0] = 0;
task_list[1] = 1;
task_list[2] = 2;
/* call this routine at task 0, 1, 2 */
LAPI_Group_create(hndl, task_num, task_list, &group);

Related information

Subroutines: LAPI_Group_free, LAPI_Xfer
LAPI_Group_free

Purpose
Marks a group handle for deallocation.

Library
Availability Library (liblapi_r.a)

C syntax

```c
#include <lapi.h>

int LAPI_Group_free(hndl, group)
    lapi_handle_t hndl;
    lapi_group_t group;
```

FORTRAN syntax

```fortran
include 'lapif.h'

lapi_group_create(hndl, group, ierror)
integer hndl
type (lapi_group_t) :: group
integer ierror
```

Parameters

Input
```
hndl
    Specifies the LAPI handle.
group
    Specifies the group handle to be deallocated.
```

Output
```
ierror
    Specifies a FORTRAN return code. This is always the last parameter.
```

Description

Type of call: collective operation (blocking)

This subroutine deallocates the specified group handle. Only the members of the group can participate in the LAPI_Group_free operation.

Return values

```
LAPI_SUCCESS
    Indicates that the function call completed successfully.
LAPI_ERR_GRP
    Indicates that the group passed in is not valid (not initialized or already deallocated).
LAPI_ERR_HNDL_INVALID
    Indicates that the hndl passed in is not valid (not initialized or in terminated state).
```

Location

```
/usr/lib/liblapi_r.a
```
LAPI_Group_free

C examples

To create a multicast group and then deallocate the group handle:

```c
lapi_handle_t hndl; /* the LAPI handle */
lapi_group_t group; /* the group handle */
int task_num; /* number of tasks in the newly-created group */
int task_list[NUM_TASKS]; /* array to store task IDs */

task_num = 3;
task_list[0] = 0;
task_list[1] = 1;
task_list[2] = 2;
/* call this routine at task 0, 1, 2 */
LAPI_Group_create(hndl, task_num, task_list, &group);

/* call this routine at task 0, 1, 2 */
LAPI_Group_free(hndl, &group);
```

Related information

Subroutines: LAPI_Group_create, LAPI_Xfer
LAPI_Init

Purpose

Initializes a LAPI context.

Library

Availability Library (liblapi_r.a)

C syntax

```c
#include <lapi.h>

int LAPI_Init(hndl, lapi_info)
    lapi_handle_t *hndl;
    lapi_info_t *lapi_info;
```

FORTRAN syntax

```fortran
include 'lapif.h'

lapi_init(hndl, lapi_info, ierror)
    integer hndl
    type (lapi_info_t) :: lapi_info
    integer ierror
```

Parameters

**Input/output**

- `lapi_info`: Specifies a structure that provides the parallel job information with which this LAPI context is associated. The value of this parameter cannot be Null (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

**Output**

- `hndl`: Specifies a pointer to the LAPI handle to initialize.
- `ierror`: Specifies a FORTRAN return code. This is always the last parameter.

Description

**Type of call**: Local initialization

Use this subroutine to instantiate and initialize a new LAPI context. A handle to the newly-created LAPI context is returned in `hndl`. All subsequent LAPI calls can use `hndl` to specify the context of the LAPI operation. Except for `LAPI_Address()` and `LAPI_Msg_string()`, the user cannot make any LAPI calls before calling `LAPI_Init()`.

The `lapi_info` structure (`lapi_info_t`) must be "zeroed out" before any fields are filled in. To do this in C, use this statement: `bzero (lapi_info, size of (lapi_info_t))`. In FORTRAN, you need to "zero out" each field manually in the `LAPI_INFO_T` type. Fields with a description of `Future support` should not be used because the names of those fields might change.
The `lapi_info_t` structure is defined as follows:

```c
typedef struct {
    lapi_dev_t protocol;
    lapi_lib_t lib_vers;
    uint epoch_num;
    int num_compl_hdlr_thr;
    uint instance_no;
    int info6;
    LAPI_err_hdlr *err_hdlr;
    com_thread_info_t *lapi_thread_attr;
    char *protocol_name;
    lapi_extend_t *add_info;
} lapi_info_t;
```

The fields are used as follows:

- **protocol**: LAPI sets this field to the protocol that has been initialized.
- **lib_vers**: Is used to indicate a library version to LAPI for compatibility purposes. Valid values for this field are:
  - `L1_LIB`: Provides the basic LAPI functions. This is the default.
  - `L2_LIB`: Provides the ability to use counters as structures.
  - `L3_LIB`: Provides additional LAPI functions.
  - `L4_LIB`: Provides additional LAPI functions.
  - `LAST_LIB`: Provides the most current level of LAPI functions. For new users of LAPI, `lib_vers` should be set to `LAST_LIB`.
- **epoch_num**: This field is no longer used.
- **num_compl_hdlr_thr**: LAPI only allows one completion handler thread, so this value is ignored.
- **instance_no**: This field is ignored.
- **info6**: This field is for future use.
- **err_hdlr**: Use this field to optionally pass a callback pointer to an error-handler routine.
- **lapi_thread_attr**: Supports thread attributes and initialization function.
- **protocol_name**: Optionally specifies the protocol (or parallel API) to initialize.
- **add_info**: Is used for additional information in standalone UDP mode.

When calling `LAPI_Init`, you can optionally set the `protocol_name` field and allocate and free the memory that is used by this field. A protocol (or parallel API) name can be any alphanumeric string and is not case sensitive. `lapi` and `mpi` are reserved parallel API names. `lapi` is used for applications that don’t specify a parallel API name. `mpi` is used for IBM MPI implementations. MPI does not use the `protocol_name` field. Instead, it passes the `LAPI_SHARE_PROTOCOL` bit flag in the `protocol` field. For compatibility, if this flag is set, LAPI automatically assumes...
the parallel API name is mpi and ignores the setting of the protocol_name field. If
you do not set the protocol_name field or the LAPI_SHARE_PROTOCOL bit flag,
LAPI assumes the parallel API name is lapi.

Multiple parallel APIs can share one LAPI context and one parallel API can use
multiple LAPI contexts. For example:

\[ \text{MP_MSG_API} = \text{"mpi,upc_lapi(8)"} \]

indicates that there is one LAPI context for mpi and that there are eight LAPI
contexts to be shared between lapi and upc.

In most cases, the order in which LAPI_Init is called with a protocol name passed
does not matter, unless you want a LAPI context to be used by a specific
protocol. In this case, you can adjust the order of protocols in the protocol
specification.

LAPI_Init fails with the error LAPI_ERR_MSG_API if either of these conditions is
met:
- A protocol name doesn’t find a match in MP_MSG_API or TWS LoadLeveler
  network keyword specification.
- The number of times that a protocol is initialized exceeds the number of LAPI
  contexts that the protocol is associated with.

When LAPI_Init is successful, an opaque handle is returned to the user. The
handle contains information for locating the corresponding LAPI context and the
protocol that’s using it.

Restrictions

When two or more parallel APIs share a single context, the second parallel API
cannot be initialized after the first parallel API has been terminated.

For systems running PE:

User space (US) mode and UDP/IP mode are supported for shared handles as long
as the same communication mode is used for both handles. Mixed transport
protocols such as LAPI IP and shared US are not supported.

Return values

<table>
<thead>
<tr>
<th>LAPI_SUCCESS</th>
<th>Indicates that the function call completed successfully.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPI_ERR_ALL_HNDL_IN_USE</td>
<td>All available LAPI instances are in use.</td>
</tr>
<tr>
<td>LAPI_ERR_BOTH_NETSTR_SET</td>
<td>Both the MP_LAPI_NETWORK and MP_LAPI_INET statements are set (only one should be set).</td>
</tr>
<tr>
<td>LAPI_ERR_CSS_LOAD_FAILED</td>
<td>LAPI is unable to load the communication utility library.</td>
</tr>
<tr>
<td>LAPI_ERR_HNDL_INVALID</td>
<td>The lapi_handle_t * passed to LAPI for initialization is Null (in C) or LAPI_ADDR_NULL (in FORTRAN).</td>
</tr>
</tbody>
</table>
LAPI_Init

LAPI_ERR_INFO_NONZERO_INFO
The future support fields in the lapi_info_t structure
that was passed to LAPI are not set to zero (and
should be).

LAPI_ERR_INFO_NULL
The lapi_info_t pointer passed to LAPI is Null (in
C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_MEMORY_EXHAUSTED
LAPI is unable to obtain memory from the system.

LAPI_ERR_MSG_API
Indicates that the MP_MSG_API environment
variable is not set correctly.

LAPI_ERR_NO_UDP_HNDLR
You passed a value of Null (in C) or
LAPI_ADDR_NULL (in FORTRAN) for both the
UDP handler and the UDP list. One of these (the
UDP handler or the UDP list) must be initialized for
standalone UDP initialization. This error is returned
in standalone UDP mode only.

LAPI_ERR_PSS_NON_ROOT
You tried to initialize the persistent subsystem
(PSS) protocol as a non-root user.

LAPI_ERR_SHMKE_NOT_LOADED
LAPI's shared memory kernel extension is not
loaded.

LAPI_ERR_SHM_SETUP
LAPI is unable to set up shared memory. This error
will be returned if LAPI_USE_SHM=only and tasks
are assigned to more than one node.

LAPI_ERR_UDP_PKT_SZ
The UDP packet size you indicated is not valid.

LAPI_ERR_UNKNOWN
An internal error has occurred.

LAPI_ERR_USER_UDP_HNDLR_FAIL
The UDP handler you passed has returned a
non-zero error code. This error is returned in
standalone UDP mode only.

Location

/usr/lib/liblapi_r.a

C examples

You need to set the MP_MSG_API environment variable before LAPI is initialized:

MP_MSG_API="api1[_api2]...[(count1)],api3[_api4]...[(count2)]..."

The following environment variables are also commonly used:

LAPI_USE_SHM=[yes|no|only] (no is the default)

MP_EUILIB=[ip|us] (ip is the default)

MP_PROCS=number_of_tasks_in_job

To initialize LAPI, follow these steps:

1. Set environment variables (as described in "Setting environment variables" on
page 39) before the user application is invoked. The remaining steps are done
in the user application.
2. Clear `lapi_info_t`, then set any fields.
3. Call `LAPI_Init`.

To initialize a LAPI handle:

```c
{l
    lapi_handle_t hndl;
    lapi_info_t info;
    bzero(&info, sizeof(lapi_info_t)); /* clear lapi_info */
    LAPI_Init(&hndl, &info);
}
```

To initialize a LAPI handle and register an error handler:

```c
void my_err_hndlr(lapi_handle_t *hndl, int *error_code, lapi_err_t *err_type,
                   int *task_id, int *src )
{
    /* examine passed parameters and delete desired information */
    if ( user wants to terminate ) {
        LAPI_Term(*hndl); /* will terminate LAPI */
        exit(some_return_code);
    }
    /* any additional processing */
    return; /* signals to LAPI that error is non-fatal; execution should continue */
}
```

```c
{l
    lapi_handle_t hndl;
    lapi_info_t info;
    bzero(&info, sizeof(lapi_info_t)); /* clear lapi_info */
    /* set error handler pointer */
    info.err_hndlr = (LAPI_err_hndlr) my_err_hndlr;
    LAPI_Init(&hndl, &info);
}
```

For standalone systems:

To initialize a LAPI handle for UDP/IP communication using a user handler:

```c
int my_udp_hndlr(lapi_handle_t *hndl, lapi_udp_t *local_addr, lapi_udp_t *addr_list,
                  lapi_udpinfo_t *info)
{
    /* LAPI will allocate and free addr_list pointer when using */
    /* a user handler */
    /* use the AIX inet_addr call to convert an IP address */
    /* from a dotted quad to a long */
    task_0_ip_as_long = inet_addr(task_0_ip_as_string);
    addr_list[0].ip_addr = task_0_ip_as_long;
    addr_list[0].port_no = task_0_port_as_unsigned;
    task_1_ip_as_long = inet_addr(task_1_ip_as_string);
    addr_list[1].ip_addr = task_1_ip_as_long;
    addr_list[1].port_no = task_1_port_as_unsigned;
```

Chapter 20. Subroutines for all systems (PE and standalone)  203
To initialize a LAPI handle for UDP/IP communication using a user list:

```c
lapi_handle_t hndl;
lapi_info_t info;
lapi_extend_t extend_info;
lapi_udp_t *addr_list;

bzero(&info, sizeof(lapi_info_t)); /* clear lapi_info */
bzero(&extend_info, sizeof(lapi_extend_t)); /* clear lapi_extend_info */

/* when using a user list, the user is responsible for allocating */
/* and freeing the list pointer */
addr_list = malloc(num_tasks);

/* Note, since we need to know the number of tasks before LAPI is */
/* initialized, we can't use LAPI_Qenv. getenv("MP_PROCS") will */
/* do the trick. */

/* populate addr_list */
/* use the AIX inet_addr call to convert an IP address */
/* from a dotted quad to a long */
task_0_ip_as_long = inet_addr(task_0_ip_as_string);
addr_list[0].ip_addr = task_0_ip_as_long;
addr_list[0].port_no = task_0_port_as_unsigned;

task_1_ip_as_long = inet_addr(task_1_ip_as_string);
addr_list[1].ip_addr = task_1_ip_as_long;
addr_list[1].port_no = task_1_port_as_unsigned;

/* then assign to extend pointer */
extend_info.add_udp_addrs = addr_list;
```

LAPI_Init
LAPI_Init

info.add_info = &extend_info;

LAPI_Init(&hndl, &info);
 .
 .
 .

/ user's responsibility only in the case of user list */
free(addr_list);

See the LAPI sample programs for complete examples of initialization in standalone mode.

To initialize a LAPI handle for user space (US) communication in standalone mode:

export MP_MSG_API=lapi
export MP_EUILIB=us
export MP_PROCS= /* number of tasks in job */
export MP_PARTITION= /* unique job key */
export MP_CHILD= /* unique task ID */
export MP_LAPI_NETWORK=01:164,sn0 /* LAPI network information */

run LAPI jobs as normal

See the README.LAPI.STANDALONE.US file in the standalone/us directory of the LAPI sample files for complete details.

Related information

- “Initializing LAPI” on page 42
- “Initializing a standalone system” on page 143
- “Bulk message transfer on AIX” on page 63

Subroutines: LAPI_Msg_string, LAPI_Term
LAPI_Msg_string

Purpose
Retrieves the message that is associated with a subroutine return code.

Library
Availability Library (liblapi_r.a)

C syntax
#include <lapi.h>
LAPI_Msg_string(error_code, buf)
int error_code;
void *buf;

FORTRAN syntax
include 'lapif.h'
lapi_msg_string(error_code, buf, ierror)
integer error_code
character buf(LAPI_MAX_ERR_STRING)
integer ierror

Parameters
Input
error_code Specifies the return value of a previous LAPI call.

Output
buf Specifies the buffer to store the message string.
ierror Specifies a FORTRAN return code. This is always the last parameter.

Description
Type of call: local queries
Use this subroutine to retrieve the message string that is associated with a LAPI return code. LAPI tries to find the messages of any return codes that come from the operating system (or for AIX, its communication subsystem).

Return values
LAPI_SUCCESS Indicates that the function call completed successfully.
LAPI_ERR_CATALOG_FAIL Indicates that the message catalog cannot be opened. An English-only string is copied into the user's message buffer (buf).
LAPI_ERR_CODE UNKNOWN Indicates that error_code is outside of the range known to LAPI.
**LAPI_Err_Return_Ptr_Null**

Indicates that the value of the `buf` pointer is Null (in C) or that the value of `buf` is `LAPI_ADDR_NULL` (in FORTRAN).

**Location**

`/usr/lib/liblapi_r.a`

**C examples**

To get the message string associated with a LAPI return code:

```c
{  
    char msg.buf[LAPI_MAX_ERR_STRING]; /* constant defined in lapi.h */  
    int rc, errc;  
    rc = some_LAPI_call();  
    errc = LAPI_Msg_string(rc, msg.buf);  
    /* msg.buf now contains the message string for the return code */}
```
LAPI_Msgpoll

LAPI_Msgpoll

Purpose

Allows the calling thread to check communication progress.

Library

Availability Library (liblapi_r.a)

C syntax

```c
#include <lapi.h>

int LAPI_Msgpoll(hndl, cnt, info)

lapi_handle_t   hndl;
uint            cnt;
lapi_msg_info_t *info;

typedef struct {
    lapi_msg_state_t status; /* Message status returned from LAPI_Msgpoll */
    ulong            reserve[10]; /* Reserved */
} lapi_msg_info_t;
```

FORTRAN syntax

```fortran
include 'lapif.h'

lapi_msgpoll(hndl, cnt, info, ierror)
nint hndl
iinteger cnt
type (LAPI_MSG_STATE_T) :: info
iinteger ierror
```

Parameters

**Input**

- **hndl**: Specifies the LAPI handle.
- **cnt**: Specifies the maximum number of times the dispatcher should loop with no progress before returning.
- **info**: Specifies a status structure that contains the result of the LAPI_Msgpoll() call. LAPI will set flags depending on the status of any send or receive completions.

**Output**

- **ierror**: Specifies a FORTRAN return code. This is always the last parameter.

Description

**Type of call**: local progress monitor (blocking)

The LAPI_Msgpoll subroutine allows the calling thread to check communication progress. With this subroutine, LAPI provides a means of running the dispatcher several times until either progress is made or a specified maximum number of dispatcher loops have executed. Here, progress is defined as the completion of either a message send operation or a message receive operation.
LAPI_Msgpoll

LAPI_Msgpoll is intended to be used when interrupts are turned off. If the user has not explicitly turned interrupts off, LAPI temporarily disables interrupt mode while in this subroutine because the dispatcher is called, which will process any pending receive operations. If the LAPI dispatcher loops for the specified maximum number of times, the call returns. If progress is made before the maximum count, the call will return immediately. In either case, LAPI will report status through a data structure that is passed by reference.

The lapi_msg_info_t structure contains a flags field (status), which is of type lapi_msg_state_t. Flags in the status field are set as follows:

- **LAPI_DISP_CNTR**
  If the dispatcher has looped \( cnt \) times without making progress

- **LAPI_SEND_COMPLETE**
  If a message send operation has completed

- **LAPI_RECV_COMPLETE**
  If a message receive operation has completed

- **LAPI_BOTH_COMPLETE**
  If both a message send operation and a message receive operation have completed

- **LAPI_POLLING_NET**
  If another thread is already polling the network or shared memory completion

**Return values**

- **LAPI_SUCCESS**
  Indicates that the function call completed successfully.

- **LAPI_ERR_HNDL_INVALID**
  Indicates that the \( hndl \) passed in is not valid (not initialized or in terminated state).

- **LAPI_ERR_MSG_INFO_NULL**
  Indicates that the \( info \) pointer is Null (in C) or that the value of \( info \) is **LAPI_ADDR_NULL** (in FORTRAN).

**Location**

/user/lib/liblapi_r.a

**C examples**

To loop through the dispatcher no more than 1000 times, then check what progress has been made:

```c
lapi_msg_info_t msg_info;
int cnt = 1000;
.
.
LAPI_Msgpoll(hndl, cnt, &msg_info);

if ( msg_info.status & LAPI_BOTH_COMPLETE ) {
    /* both a message receive and a message send have been completed */
} else if ( msg_info.status & LAPI_RECV_COMPLETE ) {
    /* just a message receive has been completed */
} else if ( msg_info.status & LAPI_SEND_COMPLETE ) {
    /* just a message send has been completed */
} else {
    /* cnt loops and no progress */
```
Related information

Subroutines: LAPI_Getcntr, LAPI_Probe, LAPI_Setcntr, LAPI_Waitcntr
LAPI_Probe

Purpose

Transfers control to the communication subsystem to check for arriving messages and to make progress in polling mode.

Library

Availability Library (liblapi_r.a)

C syntax

```c
#include <lapi.h>
int LAPI_Probe(hndl);
lapi_handle_t hndl;
```

FORTRAN syntax

```fortran
include 'lapif.h'
lapi_probe(hndl, ierror)
 integer hndl
 integer ierror
```

Parameters

Input

*hndl* Specifies the LAPI handle.

Output

*ierror* Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: local progress monitor (non-blocking)

This subroutine transfers control to the communication subsystem in order to make progress on messages associated with the context *hndl*. A LAPI_Probe operation lasts for one round of the communication dispatcher.

Note: There is no guarantee about receipt of messages on the return from this function.

Return values

- **LAPI_SUCCESS** Indicates that the function call completed successfully.
- **LAPI_ERR_HNDL_INVALID** Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

Location

/usr/lib/liblapi_r.a
LAPI_Probe

Related information
Subroutines: LAPI_Getcntr, LAPI_Msgpoll, LAPI_Nopoll_wait, LAPI_Waitcntr
LAPI_Put

Purpose

Transfers data from a local task to a remote task.

Library

Availability Library (liblapi_r.a)

C syntax

```c
#include <lapi.h>

int LAPI_Put(lapi_handle_t hndl, uint tgt, ulong len, void *tgt_addr, void *org_addr, lapi_cntr_t *tgt_cntr, org_cntr_t *org_cntr, cmpl_cntr_t *cmpl_cntr);
```

FORTRAN syntax

```fortran
include 'lapif.h'

lapi_put(hndl, tgt, len, tgt_addr, org_addr, tgt_cntr, org_cntr, ierror)
```

Parameters

**Input**

- **hndl**
  - Specifies the LAPI handle.

- **tgt**
  - Specifies the task ID of the target task. The value of this parameter must be in the range 0 <= tgt < NUM_TASKS.

- **len**
  - Specifies the number of bytes to be transferred. This parameter must be in the range 0 <= len <= the value of LAPI constant LAPI_MAX_MSG_SZ.

- **tgt_addr**
  - Specifies the address on the target task where data is to be copied into. If len is 0, The value of this parameter can be Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

- **org_addr**
  - Specifies the address on the origin task from which data is to be copied. If len is 0, The value of this parameter can be Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

**Input/output**

- **tgt_cntr**
  - Specifies the target counter address. The target counter is
LAPI_Put

incremented upon message completion. If this parameter is Null (in C) or LAPI_ADDR_NULL (in FORTRAN), the target counter is not updated.

org_cntr Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented at buffer availability. If this parameter is Null (in C) or LAPI_ADDR_NULL (in FORTRAN), the origin counter is not updated.

cmpl_cntr Specifies the completion counter address (in C) or the completion counter (in FORTRAN) that is a reflection of tgt_cntr. The completion counter is incremented at the origin after tgt_cntr is incremented. If this parameter is Null (in C) or LAPI_ADDR_NULL (in FORTRAN), the completion counter is not updated.

Output

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: point-to-point communication (non-blocking)

Use this subroutine to transfer data from a local (origin) task to a remote (target) task. The origin counter will increment on the origin task upon origin buffer availability. The target counter will increment on the target and the completion counter will increment at the origin task upon message completion. Because there is no completion handler, message completion and target buffer availability are the same in this case.

This is a non-blocking call. The caller cannot assume that the data transfer has completed upon the return of the function. Instead, counters should be used to ensure correct buffer accesses as defined above.

Note that a zero-byte message does not transfer data, but it does have the same semantic with respect to counters as that of any other message.

Return values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_DATA_LEN Indicates that the value of len is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in terminated state).

LAPI_ERR_ORG_ADDR_NULL Indicates that the org_addr parameter passed in is Null (in C) or LAPI_ADDR_NULL (in FORTRAN), but len is greater than 0.

LAPI_ERR_TGT Indicates that the tgt passed in is outside the range of tasks defined in the job.

LAPI_ERR_TGT_ADDR_NULL Indicates that the tgt_addr parameter passed in is Null (in C) or LAPI_ADDR_NULL (in FORTRAN), but len is greater than 0.
LAPI_Put

LAPI_ERR_TGT_PURGED  Indicates that the subroutine returned early because LAPI_Purge_totask() was called.

Location

/usr/lib/liblapi_r.a

C examples

{
    /* initialize the table buffer for the data addresses */
    /* get remote data buffer addresses */
    LAPI_Address_init(hndl,(void *)data_buffer,data_buffer_list);
    ...
    LAPI_Put(hndl, tgt, (ulong) data_len, (void *)(data_buffer_list[tgt]),
    (void *) data_buffer, tgt_cntr, org_cntr, compl_cntr);
    /* transfer data_len bytes from local address data_buffer. */
    /* write the data starting at address data_buffer_list[tgt] on */
    /* task tgt. tgt_cntr, org_cntr, and compl_cntr can be Null. */
}

Related information

Subroutines: LAPI_Get, LAPI_Getcntr, LAPI_Qenv, LAPI_Setcntr,
LAPI_Waitcntr, LAPI_Xfer
LAPI_Putv

Purpose

Transfers vectors of data from a local task to a remote task.

Library

Availability Library (liblapi_r.a)

C syntax

```c
#include <lapi.h>

int LAPI_Putv(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr, cmpl_cntr)

lapi_handle_t hndl;
uint tgt;
lapi_vec_t *tgt_vec;
lapi_vec_t *org_vec;
lapi_cntr_t *tgt_cntr;
lapi_cntr_t *org_cntr;
lapi_cntr_t *cmpl_cntr;

typedef struct {
    lapi_vectype_t vec_type; /* operation code */
    uint num_vecs; /* number of vectors */
    void **info; /* vector of information */
    ulong *len; /* vector of lengths */
} lapi_vec_t;
```

FORTRAN syntax

```fortran
include 'lapif.h'

lapi_putv(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr, cmpl_cntr, ierror)

integer hndl
integer tgt
integer (kind=LAPI_ADDR_TYPE) :: tgt_vec
type (lapi_vec_t) :: org_vec
integer (kind=LAPI_ADDR_TYPE) :: tgt_cntr
type (lapi_cntr_t) :: org_cntr
type (lapi_cntr_t) :: cmpl_cntr
integer ierror

The 32-bit version of the lapi_vec_t type is defined as:

```fortran
type lapi_vec_t
    sequence
    integer(4) :: vec_type
    integer(4) :: num_vecs
    integer(4) :: info
    integer(4) :: len
end type lapi_vec_t
```

The 64-bit version of the lapi_vec_t type is defined as:

```fortran
type lapi_vec_t
    sequence
    integer(8) :: vec_type
    integer(8) :: num_vecs
    integer(8) :: info
    integer(8) :: len
end type lapi_vec_t
```
Parameters

Input

hndl Specifies the LAPI handle.
tgt Specifies the task ID of the target task. The value of this parameter must be in the range $0 \leq tgt < \text{NUM\_TASKS}$.
tgt_vec Points to the target vector description.
org_vec Points to the origin vector description.

Input/output

tgt_cntr Specifies the target counter address. The target counter is incremented upon message completion. If this parameter is Null (in C) or LAPI\_ADDR\_NULL (in FORTRAN), the target counter is not updated.
org_cntr Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented at buffer availability. If this parameter is Null (in C) or LAPI\_ADDR\_NULL (in FORTRAN), the origin counter is not updated.
cmpl_cntr Specifies the completion counter address (in C) or the completion counter (in FORTRAN) that is a reflection of tgt_cntr. The completion counter is incremented at the origin after tgt_cntr is incremented. If this parameter is Null (in C) or LAPI\_ADDR\_NULL (in FORTRAN), the completion counter is not updated.

Output

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: point-to-point communication (non-blocking)

LAPI\_Putv is the vector version of the LAPI\_Put call. Use this subroutine to transfer vectors of data from the origin task to the target task. The origin vector descriptions and the target vector descriptions are located in the address space of the origin task. However, the values specified in the info array of the target vector must be addresses in the address space of the target task.

The calling program cannot assume that the origin buffer can be changed or that the contents of the target buffers on the target task are ready for use upon function return. After the origin counter (org_cntr) is incremented, the origin buffers can be modified by the origin task. After the target counter (tgt_cntr) is incremented, the target buffers can be modified by the target task. If you provide a completion counter (cmpl_cntr), it is incremented at the origin after the target counter (tgt_cntr) has been incremented at the target. If the values of any of the counters or counter addresses are Null (in C) or LAPI\_ADDR\_NULL (in FORTRAN), the data transfer occurs, but the corresponding counter increments do not occur.

If a strided vector is being transferred, the size of each block must not be greater than the stride size in bytes.

The length of any vector pointed to by org_vec must be equal to the length of the corresponding vector pointed to by tgt_vec.
LAPI_Putv

LAPI does not check for any overlapping regions among vectors either at the origin or the target. If the overlapping regions exist on the target side, the contents of the target buffer are undefined after the operation.

See LAPI_Amsendv for more information about using the various vector types. (LAPI_Putv does not support the LAPI_GEN_GENERIC type.)

Return values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in terminated state).

LAPI_ERR_ORG_EXTENT Indicates that the org_vec's extent (stride * num_vecs) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_ORG_STRIDE Indicates that the org_vec stride is less than block.

LAPI_ERR_ORG_VEC_ADDR Indicates that the org_vec->info[i] is Null (in C) or LAPI_ADDR_NULL (in FORTRAN), but its length (org_vec->len[i]) is not 0.

LAPI_ERR_ORG_VEC_LEN Indicates that the sum of org_vec->len is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_ORG_VEC_NULL Indicates that the org_vec is Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_ORG_VEC_TYPE Indicates that the org_vec->vec_type is not valid.

LAPI_ERR_STRIDE_ORG_VEC_ADDR_NULL Indicates that the strided vector address org_vec->info[0] is Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_STRIDE_TGT_VEC_ADDR_NULL Indicates that the strided vector address tgt_vec->info[0] is Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_TGT Indicates that the tgt passed in is outside the range of tasks defined in the job.

LAPI_ERR_TGT_EXTENT Indicates that tgt_vec's extent (stride * num_vecs) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_TGT_PURGED Indicates that the subroutine returned early because LAPI_Purge_totask() was called.

LAPI_ERR_TGT_STRIDE Indicates that the tgt_vec stride is less than block.

LAPI_ERR_TGT_VEC_ADDR Indicates that the tgt_vec->info[i] is Null (in C) or LAPI_ADDR_NULL (in FORTRAN), but its length (tgt_vec->len[i]) is not 0.

LAPI_ERR_TGT_VEC_LEN Indicates that the sum of tgt_vec->len is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.
LAPI_ERR_TGT_VEC_NULL Indicates that tgt_vec is Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_TGT_VEC_TYPE Indicates that the tgt_vec->vec_type is not valid.

LAPI_ERR_VEC_LEN.Diff Indicates that org_vec and tgt_vec have different lengths (len[]).

LAPI_ERR_VEC_NUM.Diff Indicates that org_vec and tgt_vec have different num_vecs.

LAPI_ERR_VEC_TYPE.Diff Indicates that org_vec and tgt_vec have different vector types (vec_type).

Location

/usr/lib/liblapi_r.a

C examples

To put a LAPI_GEN_IOVECTOR:

```c
{ /* retrieve a remote data buffer address for data to transfer, */ /* such as through LAPI_Address_init */

/* task that calls LAPI_Putv sets up both org_vec and tgt_vec */
org_vec->num_vecs = NUM_VECS;
org_vec->vec_type = LAPI_GEN_IOVECTOR;
org_vec->len = (unsigned long *)
malloc(NUM_VECS*sizeof(unsigned long));
org_vec->info = (void **) malloc(NUM_VECS*sizeof(void *));

/* each org_vec->info[i] gets a base address on the origin task */
/* each org_vec->len[i] gets the number of bytes to transfer */
/* from org_vec->info[i] */

tgt_vec->num_vecs = NUM_VECS;
tgt_vec->vec_type = LAPI_GEN_IOVECTOR;
tgt_vec->len = (unsigned long *)
malloc(NUM_VECS*sizeof(unsigned long));
tgt_vec->info = (void **) malloc(NUM_VECS*sizeof(void *));

/* each tgt_vec->info[i] gets a base address on the target task */
/* each tgt_vec->len[i] gets the number of bytes to write to vec->info[i] */
/* For LAPI_GEN_IOVECTOR, num_vecs, vec_type, and len must be the same */

LAPI_Putv(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr, compl_cntr);
/* tgt_cntr, org_cntr and compl_cntr can all be Null */

/* data will be transferred as follows: */
/* org_vec->len[0] bytes will be retrieved from */
/* org_vec->info[0] and written to tgt_vec->info[0] */
/* org_vec->len[1] bytes will be retrieved from */
/* org_vec->info[1] and written to tgt_vec->info[1] */
/* org_vec->len[NUM_VECS-1] bytes will be retrieved from */
/* org_vec->info[NUM_VECS-1] and written to */
/* tgt_vec->info[NUM_VECS-1] */
```

LAPI_Putv
LAPI_Putv

See the example in LAPI_Amsendv for information on other vector types.

Related information
Subroutines: LAPI_Amsendv, LAPI_Getcntr, LAPI_Getv, LAPI_Qenv,
LAPI_Setcntr, LAPI_Waitcntr, LAPI_Xfer
LA\_PQenv

**Purpose**

Used to query LAPI for runtime task information.

**Library**

Availability Library (\texttt{liblapi\_r.a})

**C syntax**

```
#include <lapif.h>

int LA\_PQenv(hndl, query, ret\_val)
  lapi\_handle\_t  hndl;
  lapi\_query\_t    query;
  int   *ret\_val;  /* ret\_val's type varies (see Additional query types) */
```

**FORTRAN syntax**

```
include 'lapif.h'

lapi\_qenv(hndl, query, ret\_val, ierror)
  integer hndl
  integer query
  integer ret\_val /* ret\_val's type varies (see Additional query types) */
  integer ierror
```

**Parameters**

**Input**

- \texttt{hndl}
  
  Specifies the LAPI handle.

- \texttt{query}
  
  Specifies the type of query you want to request. In C, the values for \texttt{query} are defined by the \texttt{lapi\_query\_t} enumeration in \texttt{lapif.h}. In FORTRAN, these values are defined explicitly in the 32-bit version and the 64-bit version of \texttt{lapif.h}.

**Output**

- \texttt{ret\_val}
  
  Specifies the reference parameter for LAPI to store as the result of the query. The value of this parameter cannot be Null (in C) or \texttt{LAPI\_ADDR\_NULL} (in FORTRAN).

- \texttt{ierror}
  
  Specifies a FORTRAN return code. This is always the last parameter.

**Description**

**Type of call:** local queries

Use this subroutine to query runtime settings and statistics from LAPI. LAPI defines a set of query types as an enumeration in \texttt{lapif.h} for C and explicitly in the 32-bit and 64-bit versions of \texttt{lapif.h} for FORTRAN.

For example, you can query the size of the table that LAPI uses for the \texttt{LAPI\_Addr\_set} subroutine using a \texttt{query} value of \texttt{LOC\_ADDRTABLE\_SZ}:

```
LA\_PQenv(hndl, LOC\_ADDRTABLE\_SZ, &ret\_val);
```
ret_val will contain the upper bound on the table index. A subsequent call to
LAPI_Addr_set (hndl, addr, addr_hndl); could then ensure that the value of
addr_hndl is between 0 and ret_val.

When used to show the size of a parameter, a comparison of values, or a range of
values, valid values for the query parameter of the LAPI_Qenv subroutine appear
in bold, underlined letters. For example:

**NUM_TASKS**

is a shorthand notation for:

**LAPI_Qenv(hndl, NUM_TASKS, ret_val)**

In C, **lapi_query_t** defines the valid types of LAPI queries:

typedef enum {
    TASK_ID=0,
    NUM_TASKS,
    MAX_UHDR_SZ,
    MAX_DATA_SZ,
    ERROR_CHK,
    TIMEOUT,
    MIN_TIMEOUT,
    MAX_TIMEOUT,
    INTERRUPT_SET,
    MAX_PORTS,
    MAX_PKT_SZ,
    NUM_REX_BUFS,
    REX_BUF_SZ,
    LOC_ADDRTBL_SZ,
    EPOCH_NUM,
    USE_THRESH,
    RCV_FIFO_SIZE,
    MAX_ATOM_SIZE,
    BUF_CP_SIZE,
    MAX_PKTS_OUT,
    ACK_THRESHOLD,
    QUERY_SHM_ENABLED,
    QUERY_SHM_NUM_TASKS,
    QUERY_SHM_TASKS,
    QUERY_STATISTICS,
    PRINT_STATISTICS,
    QUERY_SHM_STATISTICS,
    QUERY_LOCAL_SEND_STATISTICS,
    BULK_XFER,
    BULK_MIN_MSG_SIZE,
    RDMA_REMOTE_RCXT_AVAIL,
    RDMA_REMOTE_RCXT_TOTAL,
    RC_MAX_QP,
    RC_QP_IN_USE,
    RC_QP_USE_LRU,
    RC_QP_USE_LMC,
    RC_QP_INIT_SETUP,
    NETWORK_RESOURCES,
    LAST_QUERY
    } lapi_query_t;

typedef struct {

In FORTRAN, the valid types of LAPI queries are defined in lapif.h as follows:

```fortran
!Task_ID,Num_Tasks,Max_UHDR_SZ,Max_DATA_SZ,Error_CHK
!Timeout,Min_TIMEOUT,Max_TIMEOUT
!Interrupt_Set,Max_PORTS,Max_PKT_SZ,Num_REX_Bufs
!Loc_ADDRTBL_SZ,EPOCH_NUM,Use_THRESH
!Rcv_FIFO_Size,Max_ATOM_SIZE,Buf_CP_Size
!Max_PKTS_OUT,ACK_THRESHOLD,Query_Shm_Enabled
!Query_Shm_Num_Tasks,Query_Shm_Tasks
!Query_Statistics,Print_Statistics
!Query_Shm_Statistics,Query_Local_Send_Statistics
!Bulk_Xfer,Bulk_MIN_Msg_Size,
!Rdma_Remote_Rcxt_Avail, Rdma_Remote_Rcxt_Total
!Rc_Max_Qp, Rc_Qp_In_Use, Rc_Qp_USE_LRU
!Rc_Qp_USE_LMC, Rc_Qp_Init_Setup
!Network_Resources
!Last_Query
```

The user doing the query needs to pass in the address of a char pointer. Upon return, the address holds a pointer to a char string in LAPI. The user should not modify the char string.

**PRINT_STATISTICS**

When passed this query type, LAPI sends data transfer statistics to standard output. In this case, ret_val is unaffected. However, LAPI’s error checking requires that the value of ret_val is not Null (in C) or LAPI_ADDR_NULL (in FORTRAN) for all LAPI_Qenv types (including PRINT_STATISTICS).

This query type supports multiple parallel APIs. LAPI maintains separate communication statistics for each parallel API.
LAPI_Qenv

QUERY_LOCAL_SEND_STATISTICS
When passed this query type, LAPI_Qenv interprets ret_val as a pointer to type lapi_statistics_t. Upon function return, the fields of the structure contain LAPI's data transfer statistics for data transferred through intra-task local copy. The packet count will be 0.

This query type supports multiple parallel APIs. LAPI maintains separate communication statistics for each parallel API.

QUERY_SHM_STATISTICS
When passed this query type, LAPI_Qenv interprets ret_val as a pointer to type lapi_statistics_t. Upon function return, the fields of the structure contain LAPI's data transfer statistics for data transferred through shared memory.

This query type supports multiple parallel APIs. LAPI maintains separate communication statistics for each parallel API.

QUERY_SHM_TASKS
When passed this query type, LAPI_Qenv returns a list of task IDs with which this task can communicate using shared memory. ret_val must be an int * with enough space to hold NUM_TASKS integers. For each task i, if it is possible to use shared memory, ret_val[i] will contain the shared memory task ID. If it is not possible to use shared memory, ret_val[i] will contain -1.

QUERY_STATISTICS
When passed this query type, LAPI_Qenv interprets ret_val as a pointer to type lapi_statistics_t. Upon function return, the fields of the structure contain LAPI's data transfer statistics for data transferred using the user space (US) protocol or UDP/IP.

This query type supports multiple parallel APIs. LAPI maintains separate communication statistics for each parallel API.

Return values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in terminated state).

LAPI_ERR_QUERY_TYPE Indicates that the query passed in is not valid.

LAPI_ERR_RET_PTR_NULL Indicates that the value of the ret_val pointer is Null (in C) or that the value of ret_val is LAPI_ADDR_NULL (in FORTRAN).

Location

/usr/lib/liblapi_r.a
C examples

To query runtime values from LAPI:

```c
{  
    int task_id;
    lapi_statistics_t stats;
    ...
    LAPI_Qenv(hndl, TASK_ID, &task_id);
    /* task_id now contains the task ID */
    ...
    LAPI_Qenv(hndl, QUERY_STATISTICS, (int *)&stats);
    /* the fields of the stats structure are now filled in with runtime values */
    ...
}
```

Related information

Subroutines: `LAPI_Amsend`, `LAPI_Get`, `LAPI_Put`, `LAPI_Senv`, `LAPI_Xfer`
LAPI_Rmw

Purpose

Provides data synchronization primitives.

Library

Availability Library (liblapi_r.a)

C syntax

```c
#include <lapi.h>

int LAPI_Rmw(hndl, op, tgt, tgt_var, in_val, prev_tgt_val, org_cntr)

lapi_handle_t hndl;
RMW_ops_t op;
uint tgt;
int *tgt_var;
int *in_val;
int *prev_tgt_val;
lapi_cntr_t *org_cntr;
```

FORTRAN syntax

```fortran
include 'lapif.h'

lapi_rmw(hndl, op, tgt, tgt_var, in_val, prev_tgt_val, org_cntr, ierror)

integer hndl
integer op
integer tgt
integer (kind=LAPI_ADDR_TYPE) :: tgt_var
integer in_val
integer prev_tgt_val
type (lapi_cntr_t) :: org_cntr
integer ierror
```

Parameters

Input

- **hndl** Specifies the LAPI handle.
- **op** Specifies the operation to be performed. The valid operations are:
  - COMPARE_AND_SWAP
  - FETCH_AND_ADD
  - FETCH_AND_OR
  - SWAP
- **tgt** Specifies the task ID of the target task where the read-modify-write (Rmw) variable resides. The value of this parameter must be in the range \( 0 <\leq tgt <\) NUM_TASKS.
- **tgt_var** Specifies the target read-modify-write (Rmw) variable (in FORTRAN) or its address (in C). The value of this parameter cannot be Null (in C) or LAPI_ADDR_NULL (in FORTRAN).
- **in_val** Specifies the value that is passed in to the operation (op). This value cannot be Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

Input/output
**LAPI_Rmw**

**prev_tgt_val**
Specifies the location at the origin in which the previous `tgt_var` on the target task is stored before the operation (`op`) is executed. The value of this parameter can be Null (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

**org_cntr**
Specifies the origin counter address (in C) or the origin counter (in FORTRAN). If `prev_tgt_val` is set, the origin counter (`org_cntr`) is incremented when `prev_tgt_val` is returned to the origin side. If `prev_tgt_val` is not set, the origin counter (`org_cntr`) is updated after the operation (`op`) is completed at the target side.

**Output**

**ierror**
Specifies a FORTRAN return code. This is always the last parameter.

**Description**

**Type of call:** point-to-point communication (non-blocking)

Use this subroutine to synchronize two independent pieces of data, such as two tasks sharing a common data structure. The operation is performed at the target task (`tgt`) and is atomic. The operation takes an input value (`in_val`) from the origin and performs one of four operations (`op`) on a variable (`tgt_var`) at the target (`tgt`), and then replaces the target variable (`tgt_var`) with the results of the operation (`op`). The original value (`prev_tgt_val`) of the target variable (`tgt_var`) is returned to the origin.

The operations (`op`) are performed over the context referred to by `hndl`. The outcome of the execution of these calls is as if the following code was executed atomically:

```c
*prev_tgt_val = *tgt_var;
*tgt_var = f(*tgt_var, *in_val);
```

where:

- `f(a,b) = a + b` for **FETCH_AND_ADD**
- `f(a,b) = a | b` for **FETCH_AND_OR** (bitwise or)
- `f(a,b) = b` for **SWAP**

For **COMPARE_AND_SWAP**, `in_val` is treated as a pointer to an array of two integers, and the `op` is the following atomic operation:

```c
if(*tgt_var == in_val[0]) {
    *prev_tgt_val = True;
    *tgt_var = in_val[1];
} else {
    *prev_tgt_val = False;
}
```

All `LAPI_Rmw` calls are non-blocking. To test for completion, use the `LAPI_Getcntr` and `LAPI_Waitcntr` subroutines. `LAPI_Rmw` does not include a target counter (`tgt_cntr`), so `LAPI_Rmw` calls do not provide any indication of completion on the target task (`tgt`).
LAPI_Rmw

Return values

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPI_SUCCESS</td>
<td>Indicates that the function call completed successfully.</td>
</tr>
<tr>
<td>LAPI_ERR_HNDL_INVALID</td>
<td>Indicates that the hndl passed in is not valid (not initialized or in terminated state).</td>
</tr>
<tr>
<td>LAPI_ERR_IN_VAL_NULL</td>
<td>Indicates that the in_val pointer is Null (in C) or that the value of in_val is LAPI_ADDR_NULL (in FORTRAN).</td>
</tr>
<tr>
<td>LAPI_ERR_RMW_OP</td>
<td>Indicates that op is not valid.</td>
</tr>
<tr>
<td>LAPI_ERR_TGT</td>
<td>Indicates that the tgt passed in is outside the range of tasks defined in the job.</td>
</tr>
<tr>
<td>LAPI_ERR_TGT_PURGED</td>
<td>Indicates that the subroutine returned early because LAPI_Purge_totask() was called.</td>
</tr>
<tr>
<td>LAPI_ERR_TGT_VAR_NULL</td>
<td>Indicates that the tgt_var address is Null (in C) or that the value of tgt_var is LAPI_ADDR_NULL (in FORTRAN).</td>
</tr>
</tbody>
</table>

Location

/usr/lib/liblapi_r.a

C examples

1. To synchronize a data value between two tasks (with FETCH_AND_ADD):

```c
{
    int local_var;
    int *addr_list;
    /* both tasks initialize local_var to a value */
    /* local_var addresses are exchanged and stored */
    /* in addr_list (using LAPI_Address_init). */
    /* addr_list[tgt] now contains the address of */
    /* local_var on tgt */
    ...
    /* add value to local_var on some task */
    /* use LAPI to add value to local_var on remote task */
    LAPI_Rmw(hndl, FETCH_AND_ADD, tgt, addr_list[tgt],
             value, prev_tgt_val, &org_cntr);
    /* local_var on the remote task has been increased */
    /* by value. prev_tgt_val now contains the value */
    /* of local_var on remote task before the addition */
}
```

2. To synchronize a data value between two tasks (with SWAP):

```c
{
    int local_var;
    int *addr_list;
    /* local_var addresses are exchanged and stored */
    ...
3. To conditionally swap a data value (with `COMPARE_AND_SWAP`):

```c
int local_var;
int *addr_list;
in_val[2];

/* local_var addresses are exchanged and stored */
/* in addr_list (using LAPI_Address_init). */
/* addr_list[tgt] now contains the address of */
/* local_var on tgt. */
.
.
/* if local_var on remote_task is equal to comparator, */
/* assign value to local_var on remote task */

in_val[0] = comparator;
in_val[1] = value;

LAPI_Rmw(hndl, COMPARE_AND_SWAP, tgt, addr_list[tgt],
         in_val, prev_tgt_val, &org_cntr);

/* local_var on the remote task is now in_val[1] if it */
/* had previously been equal to in_val[0]. If the swap */
/* was performed, prev_tgt_val now contains True; */
/* otherwise, it contains False. */
```

**Related information**

Subroutines: `LAPI_Address_init`, `LAPI_Getcntr`, `LAPI_Qenv`, `LAPI_Rmw64`, `LAPI_Setcntr`, `LAPI_Waitcntr`, `LAPI_Xfer`
LAPI_Rmw64

Purpose

Provides data synchronization primitives for 64-bit applications.

Library

Availability Library (liblapi_r.a)

C syntax

```c
#include <lapi.h>

int LAPI_Rmw64(hndl, op, tgt, tgt_var, in_val, prev_tgt_val, org_cntr)

lapi_handle_t hndl;
RMW_ops_t op;
uint tgt;
long long *tgt_var;
long long *in_val;
long long *prev_tgt_val;
lapi_cntr_t *org_cntr;
```

FORTRAN syntax

```fortran
include 'lapif.h'

lapi_rmw64(hndl, op, tgt, tgt_var, in_val, prev_tgt_val, org_cntr, ierror)

integer hndl
integer op
integer tgt
integer (kind=LAPI_ADDR_TYPE) :: tgt_var
integer (kind=LAPI_LONG_LONG_TYPE) :: in_val, prev_tgt_val
type (lapi_cntr_t) :: org_cntr
integer ierror
```

Parameters

**Input**

- `hndl` Specifies the LAPI handle.
- `op` Specifies the operation to be performed. The valid operations are:
  - `COMPARE_AND_SWAP`
  - `FETCH_AND_ADD`
  - `FETCH_AND_OR`
  - `SWAP`
- `tgt` Specifies the task ID of the target task where the read-modify-write (Rmw64) variable resides. The value of this parameter must be in the range `0 <= tgt < NUM_TASKS`.
- `tgt_var` Specifies the target read-modify-write (Rmw64) variable (in FORTRAN) or its address (in C). The value of this parameter cannot be Null (in C) or `LAPI_ADDR_NULL` (in FORTRAN).
- `in_val` Specifies the value that is passed in to the operation (`op`). This value cannot be Null (in C) or `LAPI_ADDR_NULL` (in FORTRAN).
prev_tgt_val specifies the location at the origin in which the previous tgt_var on the target task is stored before the operation (op) is executed. The value of this parameter can be Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

org_cntr specifies the origin counter address (in C) or the origin counter (in FORTRAN). If prev_tgt_val is set, the origin counter (org_cntr) is incremented when prev_tgt_val is returned to the origin side. If prev_tgt_val is not set, the origin counter (org_cntr) is updated after the operation (op) is completed at the target side.

Output

ierror specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: point-to-point communication (non-blocking)

This subroutine is the 64-bit version of LAPI_Rmw. It is used to synchronize two independent pieces of 64-bit data, such as two tasks sharing a common data structure. The operation is performed at the target task (tgt) and is atomic. The operation takes an input value (in_val) from the origin and performs one of four operations (op) on a variable (tgt_var) at the target (tgt), and then replaces the target variable (tgt_var) with the results of the operation (op). The original value (prev_tgt_val) of the target variable (tgt_var) is returned to the origin.

The operations (op) are performed over the context referred to by hndl. The outcome of the execution of these calls is as if the following code was executed atomically:

```c
*prev_tgt_val = *tgt_var;
*tgt_var = f(*tgt_var, *in_val);
```

where:

f(a, b) = a + b for FETCH_AND_ADD

f(a, b) = a | b for FETCH_AND_OR (bitwise or)

f(a, b) = b for SWAP

For COMPARE_AND_SWAP, in_val is treated as a pointer to an array of two integers, and the op is the following atomic operation:

```c
if(*tgt_var == in_val[0]) {
    *prev_tgt_val = True;
    *tgt_var = in_val[1];
} else {
    *prev_tgt_val = False;
}
```

This subroutine can also be used on a 32-bit processor.

All LAPI_Rmw64 calls are non-blocking. To test for completion, use the LAPI_Getcntr and LAPI_Waitcntr subroutines. LAPI_Rmw64 does not include a target counter (tgt_cntr), so LAPI_Rmw64 calls do not provide any indication of completion on the target task (tgt).
Return values

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>LAPI_SUCCESS</code></td>
<td>Indicates that the function call completed successfully.</td>
</tr>
<tr>
<td><code>LAPI_ERR_HNDL_INVALID</code></td>
<td>Indicates that the <code>hndl</code> passed in is not valid (not initialized or in terminated state).</td>
</tr>
<tr>
<td><code>LAPI_ERR_IN_VAL_NULL</code></td>
<td>Indicates that the <code>in_val</code> pointer is Null (in C) or that the value of <code>in_val</code> is <code>LAPI_ADDR_NULL</code> (in FORTRAN).</td>
</tr>
<tr>
<td><code>LAPI_ERR_RMW_OP</code></td>
<td>Indicates that <code>op</code> is not valid.</td>
</tr>
<tr>
<td><code>LAPI_ERR_TGT</code></td>
<td>Indicates that the <code>tgt</code> passed in is outside the range of tasks defined in the job.</td>
</tr>
<tr>
<td><code>LAPI_ERR_TGT_PURGED</code></td>
<td>Indicates that the subroutine returned early because <code>LAPI_Purge_totask()</code> was called.</td>
</tr>
<tr>
<td><code>LAPI_ERR_TGT_VAR_NULL</code></td>
<td>Indicates that the <code>tgt_var</code> address is Null (in C) or that the value of <code>tgt_var</code> is <code>LAPI_ADDR_NULL</code> (in FORTRAN).</td>
</tr>
</tbody>
</table>

Location

`/usr/lib/liblapi_r.a`

C examples

1. To synchronize a data value between two tasks (with `FETCH_AND_ADD`):

   ```c
   
   long long local_var;
   long long *addr_list;
   
   /* both tasks initialize local_var to a value */
   
   /* local_var addresses are exchanged and stored */
   
   /* addr_list[tgt] now contains address of */
   
   /* local_var on tgt */
   
   /* add value to local_var on some task */
   
   /* use LAPI to add value to local_var on remote task */
   LAPI_Rmw64(hndl, FETCH_AND_ADD, tgt, addr_list[tgt],
   value, prev_tgt_val, &org_cntr);
   
   /* local_var on remote task has been increased */
   
   /* by value. prev_tgt_val now contains value of */
   
   /* local_var on remote task before the addition */
   
   }
   
   2. To synchronize a data value between two tasks (with `SWAP`):

   ```c
   
   long long local_var;
   long long *addr_list;
   
   /* local_var addresses are exchanged and stored */
   ```
3. To conditionally swap a data value (with COMPARE_AND_SWAP):

```c
long long local_var;
long long *addr_list;
long long in_val[2];

/* local_var addresses are exchanged and stored */
/* in addr_list (using LAPI_Address_init64). */
/* addr_list[tgt] now contains the address of */
/* local_var on tgt. */
.
.
/* if local_var on remote_task is equal to comparator, */
/* assign value to local_var on the remote task */

in_val[0] = comparator;
in_val[1] = value;

LAPI_Rmw64(hndl, COMPARE_AND_SWAP, tgt, addr_list[tgt],
           in_val, prev_tgt_val, &org_cnt); // Local var on remote task is now in_val[1] if it
/* had previously been equal to in_val[0]. If the */
/* swap was performed, prev_tgt_val now contains */
/* True; otherwise, it contains False. */
```

**Related information**

Subroutines: LAPI_Address_init64, LAPI_Getcntr, LAPI_Qenv, LAPI_Rmw, LAPI_Setcntr, LAPI_Waitcntr, LAPI_Xfer
**LAPI_Senv**

**Purpose**

Used to set a runtime variable.

**Library**

Availability Library (liblapi_r.a)

**C syntax**

```c
#include <lapif.h>

int LAPI_Senv(hndl, query, set_val)
    lapi_handle_t hndl;
    lapi_query_t query;
    int set_val;
```

**FORTRAN syntax**

```fortran
include 'lapif.h'

lapi_senv(hndl, query, set_val, ierror)
    integer hndl
    integer query
    integer set_val
    integer ierror
```

**Parameters**

**Input**

- **hndl**
  
  Specifies the LAPI handle.

- **query**
  
  Specifies the type of query that you want to set. In C, the values for `query` are defined by the `lapi_query_t` enumeration in `lapif.h`. In FORTRAN, these values are defined explicitly in the 32-bit version and the 64-bit version of `lapif.h`.

- **set_val**
  
  Specifies the integer value of the query that you want to set.

**Output**

- **ierror**
  
  Specifies a FORTRAN return code. This is always the last parameter.

**Description**

**Type of call:** local queries

Use this subroutine to set runtime attributes for a specific LAPI instance. In C, the `lapi_query_t` enumeration defines the attributes that can be set at runtime. These attributes are defined explicitly in FORTRAN. See **LAPI_Qenv** for more information.

You can use **LAPI_Senv** to set these runtime attributes: **ACK_THRESHOLD**, **ERROR_CHK**, **INTERRUPT_SET**, and **TIMEOUT**.

**Return values**

- **LAPI_SUCCESS**
  
  Indicates that the function call completed successfully.
LAPI_Senv

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPI_ERR_HNDL_INVALID</td>
<td>Indicates that the hndl passed in is not valid (not initialized or in terminated state).</td>
</tr>
<tr>
<td>LAPI_ERR_QUERY_TYPE</td>
<td>Indicates the query passed in is not valid.</td>
</tr>
<tr>
<td>LAPI_ERR_SET_VAL</td>
<td>Indicates the set_val pointer is not in valid range.</td>
</tr>
</tbody>
</table>

Location

/usr/lib/liblapi_r.a

C examples

The following values can be set using LAPI_Senv:

```c
MACRO_OPERATOR

// ACK_THRESHOLD:
int value;
LAPI_Senv(hndl, ACK_THRESHOLD, value);
/* LAPI sends packet acknowledgements (acks) in groups, waiting until */
/* ACK_THRESHOLD packets have arrived before returning a group of acks */
/* The valid range for ACK_THRESHOLD is (1 <= value <= 30) */
/* The default is 30. */

// ERROR_CHK:
boolean toggle;
LAPI_Senv(hndl, ERROR_CHK, toggle);
/* Indicates whether LAPI should perform error checking. If set, LAPI */
/* calls will perform bounds-checking on parameters. Error checking */
/* is disabled by default. */

// INTERRUPT_SET:
boolean toggle;
LAPI_Senv(hndl, INTERRUPT_SET, toggle);
/* Determines whether LAPI will respond to interrupts. If interrupts */
/* are disabled, LAPI will poll for message completion. */
/* toggle==True will enable interrupts, False will disable. */
/* Interrupts are enabled by default. */

// TIMEOUT:
int value;
LAPI_Senv(hndl, TIMEOUT, value);
/* LAPI will time out on a communication if no response is received */
/* within timeout seconds. Valid range is (10 <= timeout <= 86400). */
/* 86400 seconds = 24 hours. Default value is 900 (15 minutes). */
```

Related information

Subroutines: LAPI_Qenv
LAPI_Setcntr

Purpose

Used to set a counter to a specified value.

Library

Availability Library (liblapi_r.a)

C syntax

```c
#include <lapi.h>

int LAPI_Setcntr( hndl, cntr, val )
lapi_handle_t hndl;
lapi_cntr_t *cntr;
int val;
```

FORTRAN syntax

```fortran
include 'lapif.h'

lapi_setcntr(hndl, cntr, val, ierror)
integer hndl
type (lapi_cntr_t) :: cntr
integer val
integer ierror
```

Parameters

**Input**

*hndl* Specifies the LAPI handle.

*val* Specifies the value to which the counter needs to be set.

**Input/output**

*cntr* Specifies the address of the counter to be set (in C) or the counter structure (in FORTRAN). The value of this parameter cannot be Null (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

**Output**

*ierror* Specifies a FORTRAN return code. This is always the last parameter.

Description

**Type of call:** Local counter manipulation

This subroutine sets *cntr* to the value specified by *val*. Because the **LAPI_Getcntr/LAPI_Setcntr** sequence cannot be made atomic, you should only use **LAPI_Setcntr** when you know there will not be any competing operations.

Return values

**LAPI_SUCCESS** Indicates that the function call completed successfully.

**LAPI_ERR_CNTR_NULL** Indicates that the *cntr* value passed in is Null (in C) or **LAPI_ADDR_NULL** (in FORTRAN).
LAPI_ERR_HNDL_INVALID  Indicates that the hndl passed in is not valid (not initialized or in terminated state).

Location

/usr/lib/liblapi_r.a

C examples

To initialize a counter for use in a communication API call:

```c
#include <lapi.h>

int initial_value, expected_value, current_value;

int main() {
    lapi_cntr_t my_tgt_cntr, *tgt_cntr_array;
    lapi_handle_t hndl;

    /* Note: the code below is executed on all tasks */
    initial_value = 0;
    expected_value = 1;

    LAPI_Setcntr(hndl, &my_tgt_cntr, initial_value);
    LAPI_Address_init(hndl, &my_tgt_cntr, tgt_cntr_array);
    LAPI_Put(....., tgt_cntr_array[tgt], ....);
    LAPI_Gfence(hndl);

    /* Wait for counter to reach value */
    for (;;) {
        LAPI_Getcntr(hndl, &my_tgt_cntr, &current_value);
        if (current_value >= expected_value) {
            break; /* out of infinite loop */
        } else {
            LAPI_Probe(hndl);
        }
    }
}
```

LAPI_Setcntr
Related information
Subroutines: LAPI_Getcntr, LAPI_Waitcntr
LAPI_Term

Purpose
Terminates and cleans up a LAPI context.

Library
Availability Library (liblapi_r.a)

C syntax
#include <lapi.h>
int LAPI_Term(hndl)
    lapi_handle_t hndl;

FORTRAN syntax
include 'lapif.h'
lapi_term(hndl, ierror)
    integer hndl
    integer ierror

Parameters
Input
hndl Specifies the LAPI handle.

Output
ierror Specifies a FORTRAN return code. This is always the last parameter.

Description
Type of call: local termination

Use this subroutine to terminate the LAPI context that is specified by hndl. Any LAPI notification threads that are associated with this context are terminated. An error occurs when any LAPI calls are made using hndl after LAPI_Term is called.

A DGSP that is registered under that LAPI handle remains valid even after LAPI_Term is called on hndl.

Communication can occur within LAPI_Term, so the user program should maintain its communication data structures when calling LAPI_Term.

Return values
LAPI_SUCCESS Indicates that the function call completed successfully.
LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in terminated state).

Location
/usr/lib/liblapi_r.a
C examples

To terminate a LAPI context (represented by *hndl*):

```c
LAPI_Term(hndl);
```

Related information

Subroutines: `LAPI_Init`, `LAPI_Purge_totask`, `LAPI_Resume_totask`
LAPI_Util

Purpose

Serves as a wrapper function for such data gather/scatter operations as registration and reservation, for updating UDP port information, and for obtaining pointers to locking and signaling functions that are associated with a shared LAPI lock. Also includes setup functions for user-initiated RDMA operations, for HPS systems running RSCT LAPI for AIX.

Library

Availability Library (liblapi_r.a)

C syntax

```c
#include <lapi.h>

int LAPI_Util(hndl, util_cmd)
    lapi_handle_t hndl;
    lapi_util_t *util_cmd;
```

FORTRAN syntax

```fortran
include 'lapif.h'

lapi_util(hndl, util_cmd, ierror)
    integer hndl
    type (fortran_util_type) :: util_cmd
    integer ierror
```

Parameters

Input

`hndl` Specifies the LAPI handle.

Input/output

`util_cmd` Specifies the command type of the utility function.

Output

`ierror` Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: Data gather/scatter program (DGSP), UDP port information, lock sharing, trigger, and user-initiated RDMA utilities

This subroutine is used for several different operations, which are indicated by the command type value in the beginning of the command structure. The `lapi_util_t` structure is defined as:

```c
typedef union {
    lapi_util_type_t Util_type;
    lapi_reg_dgsp_t RegDgsp;
    lapi_dref_dgsp_t DrefDgsp;
    lapi_resv_dgsp_t ResvDgsp;
    lapi_reg_ddm_t DdmFunc;
    lapi_add_udp_port_t Udp;
    lapi_add_udp_port_ext Udp_ext;
    lapi_pack_dgsp_t PackDgsp;
} lapi_util_t;
```
The enumerated type `lapi_util_type_t` has these values:

Table 28. `lapi_util_type_t` types

<table>
<thead>
<tr>
<th>Value of <code>Util_type</code></th>
<th>Union member as interpreted by LAPI_Util</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPI_REGISTER_DGSP</td>
<td>lapi_reg_dgsp_t</td>
</tr>
<tr>
<td>LAPI_RESERVE_DGSP</td>
<td>lapi_resv_dgsp_t</td>
</tr>
<tr>
<td>LAPI_UNRESERVE_DGSP</td>
<td>lapi_dref_dgsp_t</td>
</tr>
<tr>
<td>LAPI_REG_DDM_FUNC</td>
<td>lapi_reg_ddm_t</td>
</tr>
<tr>
<td>LAPI_DGSP_PACK</td>
<td>lapi_pack_dgsp_t</td>
</tr>
<tr>
<td>LAPI_DGSP_UNPACK</td>
<td>lapi_unpack_dgsp_t</td>
</tr>
<tr>
<td>LAPI_ADD_UDP_DEST_PORT</td>
<td>lapi_add_udp_port_t (deprecated in C)</td>
</tr>
<tr>
<td>LAPI_ADD_UDP_DEST_EXT</td>
<td>lapi_add_udp_port_ext</td>
</tr>
<tr>
<td>LAPI_GET_THREAD_FUNC</td>
<td>lapi_thread_func_t</td>
</tr>
<tr>
<td>LAPI_REMOTE_CXT</td>
<td>lapi_remote_cxt_t</td>
</tr>
<tr>
<td>LAPI_XLATE_ADDRESS</td>
<td>lapi_get_pvo_t</td>
</tr>
<tr>
<td>LAPI_REGISTER_NOTIFICATION</td>
<td>lapi_rdma_notification_t</td>
</tr>
<tr>
<td>LAPI_TRIGGER_ADD</td>
<td>lapi_trigger_util_t</td>
</tr>
<tr>
<td>LAPI_TRIGGER_REMOVE</td>
<td>lapi_trigger_util_t</td>
</tr>
</tbody>
</table>

`hndl` is not checked for command type LAPI_REGISTER_DGSP, LAPI_RESERVE_DGSP, or LAPI_UNRESERVE_DGSP.

**LAPI_REGISTER_DGSP**

You can use this operation to register a LAPI DGSP that you have created. To register a LAPI DGSP, `lapi_dgsp_descr_t idgsp` must be passed in. LAPI returns a handle (`lapi_dg_handle_t dgsp_handle`) to use for all future LAPI calls. The `dgsp_handle` that is returned by a register operation is identified as a `lapi_dg_handle_t` type, which is the appropriate type for LAPI_Xfer and LAPI_Util calls that take a DGSP. This returned `dgsp_handle` is also defined to be castable to a pointer to a `lapi_dgsp_descr_t` for those situations where the LAPI user requires read-only access to information that is contained in the cached DGSP. The register operation delivers a DGSP to LAPI for use in future message send, receive, pack, and unpack operations. LAPI creates its own copy of the DGSP and protects it by reference count. All internal LAPI operations that depend on a DGSP cached in LAPI ensure the preservation of the DGSP by incrementing the reference count when they begin a dependency on the DGSP and decrementing the count when that dependency ends. A DGSP, once registered, can be used from any LAPI instance. LAPI_Term does not discard any DGSPs.

You can register a DGSP, start one or more LAPI operations using the DGSP, and then unreserve it with no concern about when the LAPI operations that depend on the DGSP will be done using it. See LAPI_RESERVE_DGSP and LAPI_UNRESERVE_DGSP for more information.
In general, the DGSP you create and pass in to the LAPI_REGISTER_DGSP call using the `dgsp` parameter is discarded after LAPI makes and caches its own copy. Because DGSP creation is complex, user errors may occur, but extensive error checking at data transfer time would hurt performance. When developing code that creates DGSPs, you can invoke extra validation at the point of registration by setting the `LAPI_VERIFY_DGSP` environment variable. LAPI_Util will return any detected errors. Any errors that exist and are not detected at registration time will cause problems during data transfer. Any errors detected during data transfer will be reported by an asynchronous error handler. A segmentation fault is one common symptom of a faulty DGSP. If multiple DGSPs are in use, the asynchronous error handler will not be able to identify which DGSP caused the error. For more information about asynchronous error handling, see LAPI_Init.

LAPI_REGISTER_DGSP uses the `lapi_reg DGSP` command structure.

<table>
<thead>
<tr>
<th>lapi_reg_dgsp_t field</th>
<th>lapi_reg_dgsp_t usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lapi_reg_dgsp_t field</code> type</td>
<td><code>lapi_reg_dgsp_t usage</code></td>
</tr>
<tr>
<td><code>Util_type</code></td>
<td><code>lapi_util_type_t</code></td>
</tr>
<tr>
<td><code>idgsp</code></td>
<td><code>lapi_dgsp_descr_t *</code></td>
</tr>
<tr>
<td><code>dgsp_handle</code></td>
<td><code>lapi dg_handle_t</code></td>
</tr>
<tr>
<td><code>in_usr_func</code></td>
<td><code>lapi_usr_fcall_t</code></td>
</tr>
<tr>
<td><code>status</code></td>
<td><code>lapi_status_t</code></td>
</tr>
</tbody>
</table>

LAPI_RESERVE_DGSP

You can use this operation to reserve a DGSP. This operation is provided because a LAPI client might cache a LAPI DGSP handle for later use. The client needs to ensure the DGSP will not be discarded before the cached handle is used. A DGSP handle, which is defined to be a pointer to a DGSP description that is already cached inside LAPI, is passed to this operation. The DGSP handle is also defined to be a structure pointer, so that client programs can get direct access to information in the DGSP. Unless the client can be certain that the DGSP will not be "unreserved" by another thread while it is being accessed, the client should bracket the access window with its own reserve/unreserve operation. The client is not to modify the cached DGSP, but LAPI has no way to enforce this. The reserve operation increments the user reference count, thus protecting the DGSP until an unreserve operation occurs. This is needed because the thread that placed the reservation will expect to be able to use or examine the cached DGSP until it makes an unreserve call (which decrements the user reference count), even if the unreserve operation that matches the original register operation occurs within this window on some other thread.

LAPI_RESERVE_DGSP uses the `lapi_resv_dgsp_t` command structure.

<table>
<thead>
<tr>
<th>lapi_resv_dgsp_t field</th>
<th>lapi_resv_dgsp_t field</th>
<th>lapi_resv_dgsp_t usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lapi_resv_dgsp_t field</code> type</td>
<td><code>lapi_resv_dgsp_t usage</code></td>
<td></td>
</tr>
<tr>
<td><code>Util_type</code></td>
<td><code>lapi_util_type_t</code></td>
<td>LAPI_RESERVE_DGSP</td>
</tr>
<tr>
<td><code>dgsp_handle</code></td>
<td><code>lapi dg_handle_t</code></td>
<td>OUT - handle for a registered DGSP program</td>
</tr>
<tr>
<td><code>in_usr_func</code></td>
<td><code>lapi_usr_fcall_t</code></td>
<td>For debugging only</td>
</tr>
</tbody>
</table>
LAPI_UNRESERVE_DGSP

You can use this operation to unregister or unreserve a DGSP. This operation decrements the user reference count. If external and internal reference counts are zero, this operation lets LAPI free the DGSP. All operations that decrement a reference count cause LAPI to check to see if the counts have both become 0 and if they have, dispose of the DGSP. Several internal LAPI activities increment and decrement a second reference count. The cached DGSP is disposable only when all activities (internal and external) that depend on it and use reference counting to preserve it have discharged their reference. The DGSP handle is passed to LAPI as a value parameter and LAPI does not Nullify the caller’s handle. It is your responsibility to not use this handle again because in doing an unreserve operation, you have indicated that you no longer count on the handle remaining valid.

LAPI_UNRESERVE_DGSP uses the lapi_dref_dgsp_t command structure.

LAPI_REG_DDM_FUNC

You can use this operation to register data distribution manager (DDM) functions. It works in conjunction with the DGSM CONTROL instruction. Primarily, it is used for MPI_Accumulate, but LAPI clients can provide any DDM function. It is also used to establish a callback function for processing data that is being scattered into a user buffer on the destination side.

The native LAPI user can install a callback without affecting the one MPI has registered for MPI_Accumulate. The function prototype for the callback function is:

```c
typedef long ddm_func_t ( /* return number of bytes processed */
    void *in, /* pointer to inbound data */
    void *inout, /* pointer to destination space */
    long bytes, /* number of bytes inbound */
    int operand, /* CONTROL operand value */
    int operation /* CONTROL operation value */
);```

A DDM function acts between the arrival of message data and the target buffer. The most common usage is to combine inbound data with data already in the target buffer. For example, if the target buffer is an array of integers and the incoming message consists of integers, the DDM function can be written to add each incoming integer to the value that is already in the buffer. The `operand` and
Operation fields of the DDM function allow one DDM function to support a range of operations with the CONTROL instruction by providing the appropriate values for these fields.

See "Using data gather/scatter programs (DGSPs)" on page 52 for more information.

LAPI_REG_DDM_FUNC uses the lapi_reg_ddm_t command structure. Each call replaces the previous function pointer, if there was one.

Table 32. The lapi_reg_ddm_t fields

<table>
<thead>
<tr>
<th>lapi_reg_ddm_t field</th>
<th>lapi_reg_ddm_t field type</th>
<th>lapi_reg_ddm_t usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Util_type</td>
<td>lapi_util_type_t</td>
<td>LAPI_REG_DDM_FUNC</td>
</tr>
<tr>
<td>ddm_func</td>
<td>ddm_func_t *</td>
<td>IN - DDM function pointer</td>
</tr>
<tr>
<td>in_usr_func</td>
<td>lapi_usr_fcall_t</td>
<td>For debugging only</td>
</tr>
<tr>
<td>status</td>
<td>lapi_status_t</td>
<td>OUT - future support</td>
</tr>
</tbody>
</table>

LAPI_DGSP_PACK

You can use this operation to gather data to a pack buffer from a user buffer under control of a DGSP. A single buffer may be packed by a series of calls. The caller provides a position value that is initialized to the starting offset within the buffer. Each pack operation adjusts position, so the next pack operation can begin where the previous pack operation ended. In general, a series of pack operations begins with position initialized to 0, but any offset is valid. There is no state carried from one pack operation to the next. Each pack operation starts at the beginning of the DGSP it is passed.

LAPI_DGSP_PACK uses the lapi_pack_dgsp_t command structure.

Table 33. The lapi_pack_dgsp_t fields

<table>
<thead>
<tr>
<th>lapi_pack_dgsp_t field</th>
<th>lapi_pack_dgsp_t field type</th>
<th>lapi_pack_dgsp_t usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Util_type</td>
<td>lapi_util_type_t</td>
<td>LAPI_DGSP_PACK</td>
</tr>
<tr>
<td>dgsp</td>
<td>lapi_dg_handle_t</td>
<td>OUT - handle for a registered DGSP program</td>
</tr>
<tr>
<td>in_buf</td>
<td>void *</td>
<td>IN - source buffer to pack</td>
</tr>
<tr>
<td>bytes</td>
<td>ulong</td>
<td>IN - number of bytes to pack</td>
</tr>
<tr>
<td>out_buf</td>
<td>void *</td>
<td>OUT - output buffer for pack</td>
</tr>
<tr>
<td>out_size</td>
<td>ulong</td>
<td>IN - output buffer size in bytes</td>
</tr>
<tr>
<td>position</td>
<td>ulong</td>
<td>IN/OUT - current buffer offset</td>
</tr>
<tr>
<td>in_usr_func</td>
<td>lapi_usr_fcall_t</td>
<td>For debugging only</td>
</tr>
<tr>
<td>status</td>
<td>lapi_status_t</td>
<td>OUT - future support</td>
</tr>
</tbody>
</table>

LAPI_DGSP_UNPACK

You can use this operation to scatter data from a packed buffer to a user buffer under control of a DGSP. A single buffer may be unpacked by a series of calls. The caller provides a position value that is initialized to the starting offset within the packed buffer. Each unpack operation adjusts position, so the next unpack operation can begin where the previous unpack operation ended. In general, a
series of unpack operations begins with position initialized to 0, but any offset is
valid. There is no state carried from one unpack operation to the next. Each unpack
operation starts at the beginning of the DGSP it is passed.

**LAPI_DGSP_UNPACK** uses the *lapi_unpack_dgsp_t* command structure.

<table>
<thead>
<tr>
<th><em>lapi_unpack_dgsp_t</em> field</th>
<th><em>lapi_unpack_dgsp_t</em> field type</th>
<th><em>lapi_unpack_dgsp_t</em> usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Utility type</td>
<td>lapi_util_type_t</td>
<td>LAPI_DGSP_UNPACK</td>
</tr>
<tr>
<td>dgsp</td>
<td>lapi_dg_handle_t</td>
<td>OUT - handle for a registered DGSP program</td>
</tr>
<tr>
<td>in_buf</td>
<td>void *</td>
<td>IN - source buffer for unpack</td>
</tr>
<tr>
<td>in_size</td>
<td>ulong</td>
<td>IN - source buffer size in bytes</td>
</tr>
<tr>
<td>out_buf</td>
<td>void *</td>
<td>OUT - output buffer for unpack</td>
</tr>
<tr>
<td>bytes</td>
<td>ulong</td>
<td>IN - number of bytes to unpack</td>
</tr>
<tr>
<td>position</td>
<td>ulong</td>
<td>IN/OUT - current buffer offset</td>
</tr>
<tr>
<td>in_usr_func</td>
<td>lapi_usr_fcall_t</td>
<td>For debugging only</td>
</tr>
<tr>
<td>status</td>
<td>lapi_status_t</td>
<td>OUT - future support</td>
</tr>
</tbody>
</table>

**LAPI_ADD_UDP_DEST_PORT**

You can use this operation in FORTRAN to update UDP port information about the
destination task. This operation can be used when you have written your own UDP
handler (*udp_hdlr*) and you need to support recovery of failed tasks. You cannot
use this operation under the POE runtime environment.

**LAPI_ADD_UDP_DEST_PORT** uses the *lapi_add_udp_port_t* command structure.

<table>
<thead>
<tr>
<th><em>lapi_add_udp_port_t</em> field</th>
<th><em>lapi_add_udp_port_t</em> field type</th>
<th><em>lapi_add_udp_port_t</em> usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Utility type</td>
<td>lapi_util_type_t</td>
<td>LAPI_ADD_UDP_DEST_PORT</td>
</tr>
<tr>
<td>tgt</td>
<td>uint</td>
<td>IN - destination task ID</td>
</tr>
<tr>
<td>udp_port</td>
<td>lapi_udp_t *</td>
<td>IN - UDP port information for the target</td>
</tr>
<tr>
<td>instance_no</td>
<td>uint</td>
<td>IN - Instance number of UDP</td>
</tr>
<tr>
<td>in_usr_func</td>
<td>lapi_usr_fcall_t</td>
<td>For debugging only</td>
</tr>
<tr>
<td>status</td>
<td>lapi_status_t</td>
<td>OUT - future support</td>
</tr>
</tbody>
</table>

**LAPI_ADD_UDP_DEST_PORT** and *lapi_add_udp_port_t* are deprecated in C.

Use **LAPI_ADD_UDP_DEST_EXT** and *lapi_add_udp_port_ext* instead.

**LAPI_ADD_UDP_DEST_EXT**

You can use this operation in C to update UDP port information about the
destination task. This operation can be used when you have written your own UDP
handler (*udp_hdlr*) and you need to support recovery of failed tasks. You cannot
use this operation under the POE runtime environment.

**LAPI_ADD_UDP_DEST_EXT** uses the *lapi_add_udp_port_ext* command structure.
Table 36. The lapi_add_udp_port_ext fields

<table>
<thead>
<tr>
<th>lapi_add_udp_port_ext field</th>
<th>lapi_add_udp_port_ext field type</th>
<th>lapi_add_udp_port_ext usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Util_type</td>
<td>lapi_util_type_t</td>
<td>LAPI_ADD_UDP_DEST_EXT</td>
</tr>
<tr>
<td>tgt</td>
<td>uint</td>
<td>IN - destination task ID</td>
</tr>
<tr>
<td>udp_port</td>
<td>lapi_udp_t *</td>
<td>IN - UDP port information for the target</td>
</tr>
<tr>
<td>instance_no</td>
<td>uint</td>
<td>IN - Instance number of UDP</td>
</tr>
<tr>
<td>in_usr_func</td>
<td>lapi_usr_fcall_t</td>
<td>For debugging only</td>
</tr>
<tr>
<td>status</td>
<td>lapi_status_t</td>
<td>OUT - future support</td>
</tr>
</tbody>
</table>

**LAPI_GET_THREAD_FUNC**

You can use this operation to retrieve various shared locking and signalling functions. Retrieval of these functions is valid only after LAPI is initialized and before LAPI is terminated. You should not call any of these functions after LAPI is terminated.

**LAPI_GET_THREAD_FUNC** uses the lapi_thread_func_t command structure.

Table 37. The lapi_thread_func_t fields

<table>
<thead>
<tr>
<th>lapi_thread_func_t field</th>
<th>lapi_thread_func_t field type</th>
<th>lapi_thread_func_t usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Util_type</td>
<td>lapi_util_type_t</td>
<td>LAPI_GET_THREAD_FUNC</td>
</tr>
<tr>
<td>mutex_lock</td>
<td>lapi_mutex_lock_t</td>
<td>OUT - mutex lock function pointer</td>
</tr>
<tr>
<td>mutex_unlock</td>
<td>lapi_mutex_unlock_t</td>
<td>OUT - mutex unlock function pointer</td>
</tr>
<tr>
<td>mutex_trylock</td>
<td>lapi_mutex_trylock_t</td>
<td>OUT - mutex try lock function pointer</td>
</tr>
<tr>
<td>mutex_getowner</td>
<td>lapi_mutex_getowner_t</td>
<td>OUT - mutex get owner function pointer</td>
</tr>
<tr>
<td>cond_wait</td>
<td>lapi_cond_wait_t</td>
<td>OUT - condition wait function pointer</td>
</tr>
<tr>
<td>cond_timedwait</td>
<td>lapi_cond_timedwait_t</td>
<td>OUT - condition timed wait function pointer</td>
</tr>
<tr>
<td>cond_signal</td>
<td>lapi_cond_signal_t</td>
<td>OUT - condition signal function pointer</td>
</tr>
<tr>
<td>cond_init</td>
<td>lapi_cond_init_t</td>
<td>OUT - initialize condition function pointer</td>
</tr>
<tr>
<td>cond_destroy</td>
<td>lapi_cond_destroy_t</td>
<td>OUT - destroy condition function pointer</td>
</tr>
</tbody>
</table>

LAPI uses the pthread library for thread ID management. You can therefore use pthread_self() to get the running thread ID and lapi_mutex_getowner_t to get the thread ID that owns the shared lock. Then, you can use pthread_equal() to see if the two are the same.

**Mutex thread functions:** **LAPI_GET_THREAD_FUNC** includes the following mutex thread functions: mutex lock, mutex unlock, mutex try lock, and mutex get owner.

**Mutex lock function pointer**

```c
int (*lapi_mutex_lock_t)(lapi_handle_t hndl);
```
This function acquires the lock that is associated with the specified LAPI handle. The call blocks if the lock is already held by another thread. Deadlock can occur if the calling thread is already holding the lock. You are responsible for preventing and detecting deadlocks.

**Parameters**

**Input**

- `hndl` Specifies the LAPI handle.

**Return values**

- 0 Indicates that the lock was acquired successfully.
- EINVAL Is returned if the lock is not valid because of an incorrect `hndl` value.

**Mutex unlock function pointer**

```c
int (*lapi_mutex_unlock_t)(lapi_handle_t hndl);
```

This function releases the lock that is associated with the specified LAPI handle. A thread should only unlock its own locks.

**Parameters**

**Input**

- `hndl` Specifies the LAPI handle.

**Return values**

- 0 Indicates that the lock was released successfully.
- EINVAL Is returned if the lock is not valid because of an incorrect `hndl` value.

**Mutex try lock function pointer**

```c
int (*lapi_mutex_trylock_t)(lapi_handle_t hndl);
```

This function tries to acquire the lock that is associated with the specified LAPI handle, but returns immediately if the lock is already held.

**Parameters**

**Input**

- `hndl` Specifies the LAPI handle.

**Return values**

- 0 Indicates that the lock was acquired successfully.
- EBUSY Indicates that the lock is being held.
- EINVAL Is returned if the lock is not valid because of an incorrect `hndl` value.

**Mutex get owner function pointer**

```c
int (*lapi_mutex_getowner_t)(lapi_handle_t hndl, pthread_t *tid);
```
This function gets the pthread ID of the thread that is currently holding the lock associated with the specified LAPI handle. \texttt{LAPI\_NULL\_THREAD\_ID} indicates that the lock is not held at the time the function is called.

\textbf{Parameters}

\textbf{Input}

\textit{hndl} \hspace{1em} Specifies the LAPI handle.

\textbf{Output}

\textit{tid} \hspace{1em} Is a pointer to hold the pthread ID to be retrieved.

\textbf{Return values}

0 \hspace{1em} Indicates that the lock owner was retrieved successfully.

\texttt{EINVAL} \hspace{1em} Is returned if the lock is not valid because of an incorrect \textit{hndl} value.

\textit{Condition functions:} \texttt{LAPI\_GET\_THREAD\_FUNC} includes the following condition functions: condition wait, condition timed wait, condition signal, initialize condition, and destroy condition.

\textbf{Condition wait function pointer}

\begin{verbatim}
int (*lapi_cond_wait_t)(lapi_handle_t hndl, lapi_cond_t *cond);
\end{verbatim}

This function waits on a condition variable (\textit{cond}). The user must hold the lock associated with the LAPI handle (\textit{hndl}) before making the call. Upon the return of the call, LAPI guarantees that the lock is still being held. The same LAPI handle must be supplied to concurrent \texttt{lapi\_cond\_wait\_t} operations on the same condition variable.

\textbf{Parameters}

\textbf{Input}

\textit{hndl} \hspace{1em} Specifies the LAPI handle.

\textit{cond} \hspace{1em} Is a pointer to the condition variable to be waited on.

\textbf{Return values}

0 \hspace{1em} Indicates that the condition variable has been signaled.

\texttt{EINVAL} \hspace{1em} Indicates that the value specified by \textit{hndl} or \textit{cond} is not valid.

\textbf{Condition timed wait function pointer}

\begin{verbatim}
int (*lapi_cond_timedwait_t)(lapi_handle_t hndl, lapi_cond_t *cond,
                           struct timespec *timeout);
\end{verbatim}

This function waits on a condition variable (\textit{cond}). The user must hold the lock associated with the LAPI handle (\textit{hndl}) before making the call. Upon the return of the call, LAPI guarantees that the lock is still being held. The same LAPI handle must be supplied to concurrent \texttt{lapi\_cond\_timedwait\_t} operations on the same condition variable.
Parameters
Input
hnsl Specifies the LAPI handle.
cond Is a pointer to the condition variable to be waited on.
timeout Is a pointer to the absolute time structure specifying the timeout.

Return values
0 Indicates that the condition variable has been signaled.
ETIMEDOUT Indicates that time specified by timeout has passed.
EINVAL Indicates that the value specified by hndl, cond, or timeout is not valid.

Condition signal function pointer
int (*lapi_cond_wait_t)(lapi_handle_t hndl, lapi_cond_t *cond);
typedef int (*lapi_cond_signal_t)(lapi_handle_t hndl, lapi_cond_t *cond);

This function signals a condition variable (cond) to wake up a thread that is blocked on the condition. If there are multiple threads waiting on the condition variable, which thread to wake up is decided randomly.

Parameters
Input
hnsl Specifies the LAPI handle.
cond Is a pointer to the condition variable to be signaled.

Return values
0 Indicates that the condition variable has been signaled.
EINVAL Indicates that the value specified by hndl or cond is not valid.

Initialize condition function pointer
int (*lapi_cond_init_t)(lapi_handle_t hndl, lapi_cond_t *cond);

This function initializes a condition variable.

Parameters
Input
hnsl Specifies the LAPI handle.
cond Is a pointer to the condition variable to be initialized.

Return values
0 Indicates that the condition variable was initialized successfully.
EAGAIN Indicates that the system lacked the necessary resources (other than memory) to initialize another condition variable.

ENOMEM Indicates that there is not enough memory to initialize the condition variable.

EINVAL Is returned if the \textit{hndl} value is not valid.

**Destroy condition function pointer**

\begin{verbatim}
int (*lapi_cond_destroy_t)(lapi_handle_t hndl, lapi_cond_t *cond);
\end{verbatim}

This function destroys a condition variable.

**Parameters**

**Input**

\begin{itemize}
  \item \textit{hndl} Specifies the LAPI handle.
  \item \textit{cond} Is a pointer to the condition variable to be destroyed.
\end{itemize}

**Return values**

\begin{itemize}
  \item \texttt{0} Indicates that the condition variable was destroyed successfully.
  \item \texttt{EBUSY} Indicates that the implementation has detected an attempt to destroy the object referenced by \textit{cond} while it is referenced (while being used in a \texttt{lapi\_cond\_wait\_t} or \texttt{lapi\_cond\_timedwait\_t} by another thread, for example).
  \item \texttt{EINVAL} Indicates that the value specified by \textit{hndl} or \textit{cond} is not valid.
\end{itemize}

**LAPI\_XLATE\_ADDRESS**

Use \texttt{LAPI\_XLATE\_ADDRESS} to pin and register user memory for use in an RDMA operation and to release memory that had previously been registered. To register memory, the address and length fields of \texttt{lapi\_get\_pvo\_t} are set to the requested address range and the operation field is set to \texttt{LAPI\_RDMA\_ACQUIRE}. The PVO that corresponds to the registered memory is returned in the \texttt{usr\_pvo} field of \texttt{lapi\_get\_pvo\_t}. Memory is deregistered by setting \texttt{usr\_pvo} to the PVO that was previously returned from the registration operation and by setting the operation field to \texttt{LAPI\_RDMA\_RELEASE}. Local memory and remote memory must be registered before an RDMA operation can be initiated.

A maximum of 16MB of unaligned blocks of virtual memory (or 32MB aligned on a 16MB boundary) can be used in a call to \texttt{LAPI\_Util} with the \texttt{LAPI\_XLATE\_ADDRESS} operation.

**LAPI\_XLATE\_ADDRESS** uses the \texttt{lapi\_get\_pvo\_t} command structure.

**Table 38. The \texttt{lapi\_get\_pvo\_t} fields**

<table>
<thead>
<tr>
<th>\texttt{lapi_get_pvo_t field}</th>
<th>\texttt{lapi_get_pvo_t field type}</th>
<th>\texttt{lapi_get_pvo_t usage}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{Util_type}</td>
<td>\texttt{lapi_util_type_t}</td>
<td>\texttt{LAPI_XLATE_ADDRESS}</td>
</tr>
<tr>
<td>\textit{length}</td>
<td>\texttt{uint}</td>
<td>\texttt{IN - address length (less than 32M)}</td>
</tr>
<tr>
<td>\textit{usr_pvo}</td>
<td>\texttt{lapi_user_pvo_t}</td>
<td>\texttt{OUT - PVO}</td>
</tr>
</tbody>
</table>
Table 38. The lapi_get_pvo_t fields (continued)

<table>
<thead>
<tr>
<th>lapi_get_pvo_t field</th>
<th>type</th>
<th>lapi_get_pvo_t usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>void *</td>
<td>IN - pointer to memory to translate</td>
</tr>
<tr>
<td>operation</td>
<td>lapi_rdma_req_t</td>
<td>Translate or free</td>
</tr>
</tbody>
</table>

**LAPI_REGISTER_NOTIFICATION**

Use **LAPI_REGISTER_NOTIFICATION** to associate a tag with a counter, a callback function, or both. This counter is updated (or callback is invoked) when the registering task is the target of a completed RDMA operation that requested target notification by specifying the registered tag. A counter is registered by setting the **flags** field of the **lapi_rdma_notification_t** structure to **LAPI_RCNTR_UPDATE**, specifying a tag value in the **rdma_tag** field, and storing the address of a LAPI counter in the **cntr** field. A callback is registered by setting the **flags** field to **LAPI_RCALLBACK**, specifying a tag value in the **rdma_tag** field, and updating the pointers for **sinfo** and **callback**. The callback function has the signature:

```c
void callback(lapi_handle_t *hndl, void *sinfo, int *src);
```

where **sinfo** is the pointer registered with the callback and **src** is the initiating task of the RDMA operation.

**LAPI_REGISTER_NOTIFICATION** uses the **lapi_rdma_notification_t** command structure.

Table 39. The lapi_rdma_notification_t fields

<table>
<thead>
<tr>
<th>lapi_rdma_notification_t field</th>
<th>lapi_rdma_notification_t field type</th>
<th>lapi_rdma_notification_t usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Util_type</td>
<td>lapi_util_type_t</td>
<td>LAPI_REGISTER_NOTIFICATION</td>
</tr>
<tr>
<td>rdma_tag</td>
<td>lapi_rdma_tag_t</td>
<td>IN - user RDMA tag</td>
</tr>
<tr>
<td>flags</td>
<td>lapi_rdma_flg_t</td>
<td>IN - type of notification</td>
</tr>
<tr>
<td>cntr</td>
<td>lapi_cntr_t</td>
<td>IN - registered counter</td>
</tr>
<tr>
<td>callback</td>
<td>remote_hndlr_t</td>
<td>IN - registered callback</td>
</tr>
<tr>
<td>sinfo</td>
<td>void *</td>
<td>IN - registered callback data</td>
</tr>
</tbody>
</table>

**LAPI_REMOTE_RCTX**

Use **LAPI_REMOTE_RCTX** to either obtain or release a remote RDMA resource known as an rCxt. A remote rCxt is obtained by setting the **dest** field of **lapi_remote_cxt_t** to the task that is to be the target of RDMA operations and by setting the **operation** field to **LAPI_RDMA_ACQUIRE**. The remote rCxt is returned in the **user_rcxt** field of the structure. The rCxt that is obtained is only valid on the task that acquired it and can only be used for an RDMA operation to the corresponding target task. The rCxt is intended to be obtained once and used for multiple RDMA operations. This RDMA resource is released by setting the **user_rcxt** field to the **rctx** value that was previously obtained by a **LAPI_RDMA_ACQUIRE** operation, the **dest** field to the owner of the RDMA resource, and the operation field to **LAPI_RDMA_RELEASE**.

**LAPI_REMOTE_RCTX** uses the **lapi_remote_cxt_t** command structure.
Table 40. The lapi_remote_cxt_t fields

<table>
<thead>
<tr>
<th>lapi_remote_cxt_t field</th>
<th>lapi_remote_cxt_t field</th>
<th>lapi_remote_cxt_t usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Util_type</td>
<td>lapi_util_type_t</td>
<td>LAPI_REMOTE_RCXT</td>
</tr>
<tr>
<td>operation</td>
<td>lapi_rdma_req_t</td>
<td>Obtain or return</td>
</tr>
<tr>
<td>dest</td>
<td>int</td>
<td>IN - the target for RDMA operations where the rCxt is used</td>
</tr>
<tr>
<td>usr_rcxt</td>
<td>lapi_user_cxt_t</td>
<td>OUT - remote rCxt</td>
</tr>
</tbody>
</table>

**LAPI_TRIGGER_ADD**
Use **LAPI_TRIGGER_ADD** to add a trigger.

**LAPI_TRIGGER_ADD** uses the **lapi_trigger_util_t** command structure.

Table 41. The lapi_trigger_util_t fields

<table>
<thead>
<tr>
<th>lapi_trigger_util_t field</th>
<th>lapi_trigger_util_t field</th>
<th>lapi_trigger_util_t usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Util_type</td>
<td>lapi_util_type_t</td>
<td>LAPI_TRIGGER_ADD</td>
</tr>
<tr>
<td>trigger</td>
<td>lapi_trigger_function_t</td>
<td>IN - trigger function pointer</td>
</tr>
<tr>
<td>trigger_name</td>
<td>char *</td>
<td>IN - name of the trigger function</td>
</tr>
</tbody>
</table>

**LAPI_TRIGGER_REMOVE**
Use **LAPI_TRIGGER_REMOVE** to remove a trigger.

**LAPI_TRIGGER_REMOVE** uses the **lapi_trigger_util_t** command structure.

Table 42. The lapi_trigger_util_t fields

<table>
<thead>
<tr>
<th>lapi_trigger_util_t field</th>
<th>lapi_trigger_util_t field</th>
<th>lapi_trigger_util_t usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Util_type</td>
<td>lapi_util_type_t</td>
<td>LAPI_TRIGGER_REMOVE</td>
</tr>
<tr>
<td>trigger</td>
<td>lapi_trigger_function_t</td>
<td>IN - trigger function pointer (ignored for LAPI_TRIGGER_REMOVE)</td>
</tr>
<tr>
<td>trigger_name</td>
<td>char *</td>
<td>IN - name of the trigger function</td>
</tr>
</tbody>
</table>

**Restrictions**
A maximum of 16MB of unaligned blocks of virtual memory (or 32MB aligned on a 16MB boundary) can be used in a call to **LAPI_Util** with the **LAPI_XLATE_ADDRESS** operation.

**Return values**

**LAPI_SUCCESS** Indicates that the function call completed successfully.

**LAPI_ERR_DATA_LEN** Indicates that the size of the data buffer is too large to be registered. This means the size is greater than 16 MB, if the buffer is not aligned on a 16 MB boundary, or greater than 32 MB, if the buffer is aligned on a 16 MB boundary (that is, the last 24 bits of the address are not 0s).
### LAPI_Util

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPI_ERR_DGSP</td>
<td>Indicates that the DGSP that was passed in is Null (in C) or LAPI_ADDR_NULL (in FORTRAN) or is not a registered DGSP.</td>
</tr>
<tr>
<td>LAPI_ERR_DGSP_ATOM</td>
<td>Indicates that the DGSP has an atom_size that is less than 0 or greater than MAX_ATOM_SIZE.</td>
</tr>
<tr>
<td>LAPI_ERR_DGSP_BRANCH</td>
<td>Indicates that the DGSP attempted a branch that fell outside of the code array. This is returned only in validation mode.</td>
</tr>
<tr>
<td>LAPI_ERR_DGSP_COPY_SZ</td>
<td>Is returned with DGSP validation turned on when MCOPY block &lt; 0 or COPY instruction with bytes &lt; 0. This is returned only in validation mode.</td>
</tr>
<tr>
<td>LAPI_ERR_DGSP_FREE</td>
<td>Indicates that LAPI tried to free a DGSP that is not valid or is no longer registered. There should be one LAPI_UNRESERVE_DGSP operation to close the LAPI_REGISTER_DGSP operation and one LAPI_UNRESERVE_DGSP operation for each LAPI_RESERVE_DGSP operation.</td>
</tr>
<tr>
<td>LAPI_ERR_DGSP_OPC</td>
<td>Indicates that the DGSP opcode is not valid. This is returned only in validation mode.</td>
</tr>
<tr>
<td>LAPI_ERR_DGSP_STACK</td>
<td>Indicates that the DGSP has a greater GOSUB depth than the allocated stack supports. Stack allocation is specified by the dgsp-&gt;depth member. This is returned only in validation mode.</td>
</tr>
<tr>
<td>LAPI_ERR_HNDL_INVALID</td>
<td>Indicates that the hndl passed in is not valid (that is, it is not initialized or it is in a terminated state).</td>
</tr>
<tr>
<td>LAPI_ERR_MEMORY_EXHAUSTED</td>
<td>Indicates that LAPI cannot obtain memory from the system or cannot allocate space to store RDMA data structures.</td>
</tr>
<tr>
<td>LAPI_ERR_NO_RDMA_RESOURCE</td>
<td>Indicates that the buffer is not read-write-addressable, that there are no more device driver resources available for pinning and mapping the buffer, or that the remote task has no more rCxts.</td>
</tr>
<tr>
<td>LAPI_ERR_UDP_PORT_INFO</td>
<td>Indicates that the udp_port information pointer is Null (in C) or that the value of udp_port is LAPI_ADDR_NULL (in FORTRAN).</td>
</tr>
<tr>
<td>LAPI_ERR_UTIL_CMD</td>
<td>Indicates that the command type is not valid.</td>
</tr>
<tr>
<td>LAPI_ERR_XLATE_FAILED</td>
<td>Translation and mapping of user memory failed.</td>
</tr>
</tbody>
</table>

### Location

/usr/lib/liblapi_r.a

### C examples

1. To create and register a DGSP:

```c
{ /*
*/
```
** DGSP code array. DGSP instructions are stored as ints (with constants defined in lapi.h for the number of integers needed to store each instruction). We will have one COPY and one ITERATE instruction in our DGSP. We use LAPI’s constants to allocate the appropriate storage. */

```
int code[LAPI_DGSM_COPY_SIZE+LAPI_DGSM_ITERATE_SIZE];
```

/* DGSP description */

```
lapi_dgsp_descr_t dgsp_d;
```

/* Data structure for the xfer call. */

```
lapi_xfer_t xfer_struct;
```

/* DGSP data structures */

```
lapi_dgsm_copy_t *copy_p; /* copy instruction */
lapi_dgsm_iterate_t *iter_p; /* iterate instruction */
int *code_ptr; /* code pointer */
```

/* constant for holding code array info */

```
int code_less_iterate_size;
```

/* used for DGSP registration */

```
lapi_reg_dgsp_t reg_util;
```

/*
 ** Set up dgsp description
 */

```
/* set pointer to code array */
dgsp_d.code = &code[0];

/* set size of code array */
dgsp_d.code_size = LAPI_DGSM_COPY_SIZE + LAPI_DGSM_ITERATE_SIZE;

/* not using DGSP gosub instruction */
dgsp_d.depth = 1;

/*
 ** set density to show internal gaps in the DGSP data layout */
dgsp_d.density = LAPI_DGSM_SPARSE;

/* transfer 4 bytes at a time */
dgsp_d.size = 4;

/* advance the template by 8 for each iteration */
dgsp_d.extent = 8;

/*
 ** ext specifies the memory 'footprint' of data to be transferred. The lex specifies the offset from the base address to begin viewing the data. The rex specifies the length from the base address to use. */
dgsp_d.lext = 0;
dgsp_d.rext = 4;
/* atom size of 0 lets LAPI choose the packet size */
dgsp_d.atom_size = 0;
```
** set up the copy instruction
```c
/*
copy_p = (lapi_dgsm_copy_t *) (dgsp_d.code);
copy_p->opcode = LAPI_DGSM_COPY;

/* copy 4 bytes at a time */
copy_p->bytes = (long) 4;

/* start at offset 0 */
copy_p->offset = (long) 0;

/* set code pointer to address of iterate instruction */
code_less_iterate_size = dgsp_d.code_size - LAPI_DGSM_ITERATE_SIZE;
code_ptr = ((int *) (code)) + code_less_iterate_size;
```

** Set up iterate instruction
```c
/*
iter_p = (lapi_dgsm_iterate_t *) code_ptr;
iter_p->opcode = LAPI_DGSM_ITERATE;
iter_p->iter_loc = (-code_less_iterate_size);

/* Set up and do DGSP registration */
reg_util.Util_type = LAPI_REGISTER_DGSP;
reg_util.idgsp = &dgsp_d;
LAPI_Util(hndl, (lapi_util_t *)&reg_util);
```

** LAPI returns a usable DGSP handle in
** reg_util.dgsp_handle
** Use this handle for subsequent reserve/unreserve
** and Xfer calls. On the receive side, this
** handle can be returned by the header handler using the
** return_info_t mechanism. The DGSP will then be used for
** scattering data.
```c
/*
*/
```

2. To reserve a DGSP handle:
```c
{

reg_util.dgsp_handle = dgsp_handle;

/*
** dgsp_handle has already been created and
** registered as in the above example
*/

reg_util.Util_type = LAPI_RESERVE_DGSP;
LAPI_Util(hndl, (lapi_util_t *)&reg_util);

/*
** LAPI's internal reference count to dgsp_handle
** will be incremented. DGSP will
** remain available until an unreserve is
** done for each reserve, plus one more for
** the original registration.
*/
```

3. To unreserve a DGSP handle:
```c
{
    reg_util.dgsp_handle = dgsp_handle;

    /*
    ** dgsp_handle has already created and
    ** registered as in the above example, and
    ** this thread no longer needs it.
    */

    reg_util.Util_type = LAPI_UNRESERVE_DGSP;
    LAPI_Util(hndl, (lapi_util_t *)&reg_util);

    /*
    ** An unreserve is required for each reserve,
    ** plus one more for the original registration.
    */
}

4. For an example of how to create a simple RDMA get/put function, see "A sample RDMA program" on page 67.

Related information
Subroutines: LAPI_Init, LAPI_Xfer
LAPI_Waitcntr

Purpose

Waits until a specified counter reaches the value specified.

Library

Availability Library (liblapi_r.a)

C syntax

```c
#include <lapi.h>

int LAPI_Waitcntr(hndl, cntr, val, cur_cntr_val)
    lapi_handle_t  hndl;
    lapi_cntr_t   *cntr;
    int            val;
    int            *cur_cntr_val;
```

FORTRAN syntax

```fortran
include 'lapif.h'

lapi_waitcntr(hndl, cntr, val, cur_cntr_val, ierror)
    integer hndl
    type (lapi_cntr_t) :: cntr
    integer val
    integer cur_cntr_val
    integer ierror
```

Parameters

**Input**

- **hndl** Specifies the LAPI handle.
- **val** Specifies the value the counter needs to reach.

**Input/output**

- **cntr** Specifies the counter structure (in FORTRAN) to be waited on or its address (in C). The value of this parameter cannot be Null (in C) or LAPI_ADDR_NULL (in FORTRAN).

**Output**

- **cur_cntr_val** Specifies the integer value of the current counter. This value can be Null (in C) or LAPI_ADDR_NULL (in FORTRAN).
- **ierror** Specifies a FORTRAN return code. This is always the last parameter.

Description

**Type of call**: local progress monitor (blocking)

This subroutine waits until `cntr` reaches or exceeds the specified `val`. Once `cntr` reaches `val`, `cntr` is decremented by the value of `val`. In this case, "decremented" is used (as opposed to "set to zero") because `cntr` could have contained a value that was greater than the specified `val` when the call was made. This call may or may not check for message arrivals over the LAPI context `hndl`. The `cur_cntr_val` variable is set to the current counter value.
LAPI_Waitcntr

Return values

LAPI_SUCCESS Indicates that the function call completed successfully.
LAPI_ERR_CNTR_NULL Indicates that the \textit{cntr} pointer is Null (in C) or that the value of \textit{cntr} is LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_HNDL_INVALID Indicates that the \textit{hndl} passed in is not valid (not initialized or in terminated state).

Location

/usr/lib/liblapi_r.a

C examples

To wait on a counter to reach a specified value:

\begin{verbatim}
{
    int val;
    int cur_cntr_val;
    lapi_cntr_t some_cntr;
    .
    .
    LAPI_Waitcntr(hndl, &some_cntr, val, &cur_cntr_val);
    /* Upon return, some_cntr has reached val */
}
\end{verbatim}

Related information

Subroutines: LAPI_Amsend, LAPI_Amsendv, LAPI_Get, LAPI_Getcntr, LAPI_Getv, LAPI_Put, LAPI_Putv, LAPI_Rmw, LAPI_Rmw64, LAPI_Setcntr, LAPI_Xfer
LAPI_Xfer

Purpose

Serves as a wrapper function for LAPI operations that are related to data transfer.

Library

Availability Library (liblapi_r.a)

C syntax

```c
#include <lapi.h>

int LAPI_Xfer(hndl, xfer_cmd)
    lapi_handle_t *hndl;
    lapi_xfer_t *xfer_cmd;

typedef struct {
    uint src; /* Target task ID */
    uint reason; /* LAPI return codes */
    ulong reserve[6]; /* Reserved */
} lapi_sh_info_t;

typedef void (scompl_hndlr_t)(lapi_handle_t *hndl, void *completion_param,
                             lapi_sh_info_t *info);
```

FORTRAN syntax

```fortran
include 'lapif.h'

lapi_xfer(hndl, xfer_cmd, ierror)
    integer hndl
    type (fortran_xfer_type) :: xfer_cmd
    integer ierror
```

Parameters

Input

- `hndl` Specifies the LAPI handle.
- `xfer_cmd` Specifies the name and parameters of the data transfer function.

Output

- `ierror` Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: point-to-point communication (non-blocking)

The LAPI_Xfer subroutine provides a superset of the functionality of these subroutines: LAPI_Amsend, LAPI_Amsendv, LAPI_Get, LAPI_Getv, LAPI_Put, LAPI_Putv, and LAPI_Rmw. In addition, LAPI_Xfer provides DGSP and multicasting functions, a low-latency transfer function for small messages, and user-initiated RDMA transfer functions (for HPS systems running RSCT LAPI for AIX).
In C, the `LAPI_Xfer` subroutine is passed a pointer to a union. It examines the first member of the union, `Xfer_type`, to determine the transfer type, and to determine which union member was passed. `LAPI_Xfer` expects every field of the identified union member to be set. It does not examine or modify any memory outside of the identified union member. `LAPI_Xfer` treats all union members (except `status`) as read-only data.

The `lapi_xfer_t` structure is defined as:

```c
typedef union {
    lapi_xfer_type_t Xfer_type;
    lapi_get_t Get;
    lapi_am_t Am;
    lapi_rmw_t Rmw;
    lapi_put_t Put;
    lapi_getv_t Getv;
    lapi_putv_t Putv;
    lapi_amv_t Amv;
    lapi_amdgsp_t Dgsp;
    lapi_hwxfer_t HwXfer;
    lapi_amx_t Amx;
    lapi_mc_t Mc;
} lapi_xfer_t;
```

Though the `lapi_xfer_t` structure applies only to the C version of `LAPI_Xfer`, the following tables include the FORTRAN equivalents of the C datatypes.

Table 43 list the values of the `lapi_xfer_type_t` structure for C and the explicit `Xfer_type` values for FORTRAN.

<table>
<thead>
<tr>
<th>Value of <code>Xfer_type</code> (C or FORTRAN)</th>
<th>Shorthand notation</th>
<th>Union member as interpreted by <code>LAPI_Xfer</code> (C)</th>
<th>Value of <code>fortran_xfer_type</code> (FORTRAN)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>LAPI_AM_XFER</code></td>
<td>AM</td>
<td>lapi_am_t</td>
<td><code>LAPI_AM_T</code></td>
</tr>
<tr>
<td><code>LAPI_AM_LW_XFER</code></td>
<td>AM_LW</td>
<td>lapi_am_t</td>
<td><code>LAPI_AM_T</code></td>
</tr>
<tr>
<td><code>LAPI_AMV_XFER</code></td>
<td>AMV</td>
<td>lapi_amv_t</td>
<td><code>LAPI_AMV_T</code></td>
</tr>
<tr>
<td><code>LAPI_AMX_XFER</code></td>
<td>AMX</td>
<td>lapi_amx_t</td>
<td><code>LAPI_AMX_T</code></td>
</tr>
<tr>
<td><code>LAPI_DGSP_XFER</code></td>
<td>DGSP</td>
<td>lapi_amdgsp_t</td>
<td><code>LAPI_AMDGSP_T</code></td>
</tr>
<tr>
<td><code>LAPI_GET_XFER</code></td>
<td>GET</td>
<td>lapi_get_t</td>
<td><code>LAPI_GET_T</code></td>
</tr>
<tr>
<td><code>LAPI_GETV_XFER</code></td>
<td>GETV</td>
<td>lapi_getv_t</td>
<td><code>LAPI_GETV_T</code></td>
</tr>
<tr>
<td><code>LAPI_MC_XFER</code></td>
<td>MC</td>
<td>lapi_mc_t</td>
<td><code>LAPI_MC_T</code></td>
</tr>
<tr>
<td><code>LAPI_PUT_XFER</code></td>
<td>PUT</td>
<td>lapi_put_t</td>
<td><code>LAPI_PUT_T</code></td>
</tr>
<tr>
<td><code>LAPI_PUTV_XFER</code></td>
<td>PUTV</td>
<td>lapi_putv_t</td>
<td><code>LAPI_PUTV_T</code></td>
</tr>
<tr>
<td><code>LAPI_RDMA_XFER</code></td>
<td>RDMA</td>
<td>lapi_hwxfer_t</td>
<td><code>LAPI_HWXFER_T</code></td>
</tr>
<tr>
<td><code>LAPI_RMW_XFER</code></td>
<td>RMW</td>
<td>lapi_rmw_t</td>
<td><code>LAPI_RMW_T</code></td>
</tr>
</tbody>
</table>

**lapi_am_t details**

`LAPI_AM_XFER`: Table 44 on page 262 shows the correspondence among the parameters of the `LAPI_Amsend` subroutine, the fields of the C `lapi_am_t` structure and their datatypes, and the equivalent FORTRAN datatypes for `LAPI_AM_XFER`. The `lapi_am_t` fields are listed in Table 44 on page 262 in the order that they occur in the `lapi_xfer_t` structure.
### LAPI_Xfer

Table 44. LAPI_Amsend and lapi_am_t equivalents

<table>
<thead>
<tr>
<th>lapi_am_t field name (C)</th>
<th>lapi_am_t field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>Equivalent LAPI_Amsend parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xfer_type</td>
<td>lapi_xfer_type_t</td>
<td>integer(4)</td>
<td>implicit in C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_Xfer value in FORTRAN:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_AM_XFER</td>
</tr>
<tr>
<td>flags</td>
<td>int</td>
<td>integer(4)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_Xfer parameter in FORTRAN:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>flags</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See &quot;Setting the flags field&quot; on</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>page 273 for more information.</td>
</tr>
<tr>
<td>tgt</td>
<td>uint</td>
<td>integer(4)</td>
<td>tgt</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_Xfer parameter in FORTRAN:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>pad</td>
</tr>
<tr>
<td>hdr_hdl</td>
<td>lapi_long_t</td>
<td>integer(8)</td>
<td>hdr_hdl</td>
</tr>
<tr>
<td>uhdr_len</td>
<td>uint</td>
<td>integer(4)</td>
<td>uhdr_len (multiple of processor’s word size)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_Xfer parameter in FORTRAN:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>pad2</td>
</tr>
<tr>
<td>uhdr</td>
<td>void *</td>
<td>integer(4) (32-bit)</td>
<td>uhdr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>udata</td>
<td>void *</td>
<td>integer(4) (32-bit)</td>
<td>udata</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>udata_len</td>
<td>ulong</td>
<td>integer(4) (32-bit)</td>
<td>udata_len</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>shdlr</td>
<td>scompl_hndlr_t *</td>
<td>integer(4) (32-bit)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td>LAPI_Xfer parameter in FORTRAN:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>shdlr</td>
</tr>
<tr>
<td>sinfo</td>
<td>void *</td>
<td>integer(4) (32-bit)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td>LAPI_Xfer parameter in FORTRAN:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sinfo</td>
</tr>
<tr>
<td>tgt_cntr</td>
<td>lapi_long_t</td>
<td>integer(8)</td>
<td>tgt_cntr</td>
</tr>
<tr>
<td>org_cntr</td>
<td>lapi_cntr_t *</td>
<td>integer(4) (32-bit)</td>
<td>org_cntr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>cmpl_cntr</td>
<td>lapi_cntr_t *</td>
<td>integer(4) (32-bit)</td>
<td>cmpl_cntr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
</tbody>
</table>

When the origin data buffer is free to be used, the pointer to the send completion handler (shdlr) is called with the send completion data (sinfo) if shdlr is not a Null pointer (in C) or LAPI_ADDR_NULL (in FORTRAN). Otherwise, the behavior is identical to that of LAPI_Amsend.

**lapi_amdgsp_t details:** Table 45 on page 263 shows the correspondence among the fields of the C lapi_amdgsp_t structure and their datatypes, how they are used in LAPI_Xfer, and the equivalent FORTRAN datatypes. The lapi_amdgsp_t fields are listed in Table 45 on page 263 in the order that they occur in the lapi_xfer_t structure.
Table 45. The lapi_amdgsp_t fields

<table>
<thead>
<tr>
<th>lapi_amdgsp_t field name (C)</th>
<th>lapi_amdgsp_t field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>LAPI_Xfer usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Xfer_type</strong></td>
<td>lapi_xfer_type_t</td>
<td>integer(4)</td>
<td>LAPI_DGSP_XFER (DGSP)</td>
</tr>
<tr>
<td><strong>flags</strong></td>
<td>int</td>
<td>integer(4)</td>
<td>Use this field to specify directives or hints to LAPI. If you do not want to use any directives or hints, set this field to 0. See &quot;Setting the flags field&quot; on page 273 for more information.</td>
</tr>
<tr>
<td><strong>tgt</strong></td>
<td>uint</td>
<td>integer(4)</td>
<td>target task</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>integer(4)</td>
<td>pad (padding alignment for FORTRAN only)</td>
</tr>
<tr>
<td><strong>hdr_hdl</strong></td>
<td>lapi_long_t</td>
<td>integer(8)</td>
<td>header handler to invoke at target</td>
</tr>
<tr>
<td><strong>uhdr_len</strong></td>
<td>uint</td>
<td>integer(4)</td>
<td>user header length (multiple of processor’s word size)</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>integer(4)</td>
<td>pad2 (padding alignment for 64-bit FORTRAN only)</td>
</tr>
<tr>
<td><strong>uhdr</strong></td>
<td>void *</td>
<td>integer(4) (32-bit)</td>
<td>pointer to user header</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td><strong>udata</strong></td>
<td>void *</td>
<td>integer(4) (32-bit)</td>
<td>pointer to user data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td><strong>udata_len</strong></td>
<td>ulong</td>
<td>integer(4) (32-bit)</td>
<td>user data length</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td><strong>shdlr</strong></td>
<td>scompl_hndlr_t *</td>
<td>integer(4) (32-bit)</td>
<td>send completion handler (optional)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td><strong>sinfo</strong></td>
<td>void *</td>
<td>integer(4) (32-bit)</td>
<td>data pointer to pass to send completion handler (optional)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td><strong>tgt_cntr</strong></td>
<td>lapi_long_t</td>
<td>integer(8)</td>
<td>target counter (optional)</td>
</tr>
<tr>
<td><strong>org_cntr</strong></td>
<td>lapi_cntr_t *</td>
<td>integer(4) (32-bit)</td>
<td>origin counter (optional)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td><strong>cmpl_cntr</strong></td>
<td>lapi_cntr_t *</td>
<td>integer(4) (32-bit)</td>
<td>completion counter (optional)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td><strong>dgsp</strong></td>
<td>lapi_dg_handle_t</td>
<td>integer(4) (32-bit)</td>
<td>Handle of a registered DGSP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td><strong>status</strong></td>
<td>lapi_status_t</td>
<td>integer(4) (32-bit)</td>
<td>Status to return (future use)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>integer(4)</td>
<td>pad3 (padding alignment for 64-bit FORTRAN only)</td>
</tr>
</tbody>
</table>

When the origin data buffer is free to be modified, the send completion handler (shdlr) is called with the send completion data (sinfo), if shdlr is not a Null pointer (in C) or LAPI_ADDR_NULL (in FORTRAN).

See "Using lapi_am_dgsp_t for scatter-side DGSP processing" for more information.

Using lapi_am_dgsp_t for scatter-side DGSP processing: LAPI allows additional information to be returned from the header handler through the use of the lapi_return_info_t datatype. For more information, see "The enhanced header"
In the case of transfer type lapi_am_dgsp_t, this mechanism can be used to instruct LAPI to run a user DGSP to scatter data on the receiving side.

To use this mechanism, pass a lapi_return_info_t * pointer back to LAPI through the msg_len member of the user header handler. The dgsp_handle member of the passed structure must point to a DGSP description that has been registered on the receiving side. See LAPI_Util and "Using data gather/scatter programs (DGSPs)" for details on building and registering DGSPs.

LAPI_AM_LW_XFER: Table 46 shows the correspondence among the fields of the C laapi_am_t structure and their datatypes, how they are used in LAPI_Xfer, and the equivalent FORTRAN datatypes for LAPI_AM_LW_XFER. The laapi_am_t fields are listed in Table 46 in the order that they occur in the laapi_xfer_t structure.

<table>
<thead>
<tr>
<th>laapi_am_t field name (C)</th>
<th>laapi_am_t field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>LAPI_Xfer usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xfer_type</td>
<td>laapi_xfer_type_t</td>
<td>integer(4)</td>
<td>LAPI_AM_LW_XFER</td>
</tr>
<tr>
<td>flags</td>
<td>int</td>
<td>integer(4)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_Xfer parameter in FORTRAN: flags</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See &quot;Setting the flags field&quot; on page 273 for more information.</td>
</tr>
<tr>
<td>tgt</td>
<td>uint</td>
<td>integer(4)</td>
<td>tgt</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>integer(4)</td>
<td>LAPI_Xfer parameter in FORTRAN: pad</td>
</tr>
<tr>
<td>hdr_hdl</td>
<td>laapi_long_t</td>
<td>integer(8)</td>
<td>hdr_hdl</td>
</tr>
<tr>
<td>uhdr_len</td>
<td>uint</td>
<td>integer(4)</td>
<td>uhdr_len (multiple of processor’s word size)</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>integer(4)</td>
<td>LAPI_Xfer parameter in FORTRAN (64-bit): pad2</td>
</tr>
<tr>
<td>uhdr</td>
<td>void *</td>
<td>integer(4) (32-bit)</td>
<td>uhdr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>udata</td>
<td>void *</td>
<td>integer(4) (32-bit)</td>
<td>udata</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>udata_len</td>
<td>ulong</td>
<td>integer(4) (32-bit)</td>
<td>udata_len</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
</tbody>
</table>

Restrictions for AM_LW follow.

On the sending side:
- The sum of uhdr_len and udata_len in laapi_am_t must not exceed 128 bytes.
- The function does not support the use of any counters. Any special counters are ignored.
- There is no send completion handler support and LAPI guarantees that buffers for user header and user data can be reused right after the function returns.
- hdr_hdl must be an index to the function table, as defined by LAPI_Addr_set.

On the receiving side, the LAPI receive function only updates the msg_len, src, and udata_one_pkt_ptr fields in the laapi_return_info_t structure before calling the
user’s header handler. The user must consume the data pointed by *udata_one_pkt_ptr* in the header handler and LAPI will *not* copy the data for the user. The return value of the header handler is always ignored. Unlike AM, using AM_LW guarantees that the receive completion handler, if provided by the user, is always executed inline. (For AM, you must set *ret_flags* to LAPI_LOCAL_STATE or LAPI_SEND_REPLY for the completion handler to be executed inline.) The following *lapi_return_info_t* fields are not supported: *ret_flags*, *ctl_flags*, and *dsp_handle*. For more information about *lapi_return_info_t*, see “The enhanced header handler interface” on page 95.

**lapi_amv_t details**

Table 47 shows the correspondence among the parameters of the LAPI_Amsendv subroutine, the fields of the C *lapi_amv_t* structure and their datatypes, and the equivalent FORTRAN datatypes. The *lapi_amv_t* fields are listed in Table 47 in the order that they occur in the *lapi_xfer_t* structure.

<table>
<thead>
<tr>
<th><em>lapi_amv_t</em> field name (C)</th>
<th><em>lapi_amv_t</em> field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>Equivalent LAPI_Amsendv parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xfer_type</td>
<td><em>lapi_xfer_type_t</em></td>
<td>integer(4)</td>
<td>implicit in C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_Xfer value in FORTRAN:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_AMV_XFER</td>
</tr>
<tr>
<td>flags</td>
<td>int</td>
<td>integer(4)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_Xfer parameter in FORTRAN:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>flags</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See &quot;Setting the flags field&quot; on page 273 for more information.</td>
</tr>
<tr>
<td>tgt</td>
<td>uint</td>
<td>integer(4)</td>
<td>tgt</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_Xfer parameter in FORTRAN:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>pad</td>
</tr>
<tr>
<td>hdr_hdl</td>
<td><em>lapi_long_t</em></td>
<td>integer(8)</td>
<td>hdr_hdl</td>
</tr>
<tr>
<td>uhdr_len</td>
<td>uint</td>
<td>integer(4)</td>
<td>uhdr_len (multiple of processor’s word size)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_Xfer parameter in FORTRAN:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>pad2</td>
</tr>
<tr>
<td>uhdr</td>
<td>void *</td>
<td>integer(4) (32-bit)</td>
<td>uhdr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>shdlr</td>
<td><em>scompl_hndlr_t</em></td>
<td>integer(4) (32-bit)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td>LAPI_Xfer parameter in FORTRAN:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>shdlr</td>
</tr>
<tr>
<td>sinfo</td>
<td>void *</td>
<td>integer(4) (32-bit)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td>LAPI_Xfer parameter in FORTRAN:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sinfo</td>
</tr>
<tr>
<td>org_vec</td>
<td><em>lapi_vec_t</em></td>
<td>integer(4) (32-bit)</td>
<td>org_vec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>tgt_cntr</td>
<td><em>lapi_long_t</em></td>
<td>integer(8)</td>
<td>tgt_cntr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_Xfer parameter in FORTRAN:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>pad2</td>
</tr>
</tbody>
</table>
Table 47. LAPI_Amsendv and lapi_amv_t equivalents (continued)

<table>
<thead>
<tr>
<th>lapi_amv_t field name (C)</th>
<th>lapi_amv_t field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>Equivalent LAPI_Amsendv parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>org_cntr</td>
<td>lapi_cntr_t *</td>
<td>integer(4) (32-bit)</td>
<td>org_cntr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>cmpl_cntr</td>
<td>lapi_cntr_t *</td>
<td>integer(4) (32-bit)</td>
<td>cmpl_cntr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
</tbody>
</table>

lapi_amx_t details
See “Extended user header support” on page 23 for additional information related to this transfer type.

Table 48 shows the correspondence among the fields of the C lapi_amx_t structure and their datatypes, how they are used in LAPI_Xfer, and the equivalent FORTRAN datatypes. The lapi_amx_t fields are listed in Table 48 in the order that they occur in the lapi_xfer_t structure.

Table 48. The lapi_amx_t fields

<table>
<thead>
<tr>
<th>lapi_amx_t field name (C)</th>
<th>lapi_amx_t field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>LAPI_Xfer usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xfer_type</td>
<td>lapi_xfer_type_t</td>
<td>integer(4)</td>
<td>LAPI_AMX_XFER (AMX)</td>
</tr>
<tr>
<td>flags</td>
<td>int</td>
<td>integer(4)</td>
<td>Use this field to specify directives or hints to LAPI. If you do not want to use any directives or hints, set this field to 0. See “Setting the flags field” on page 273 for more information.</td>
</tr>
<tr>
<td>tgt</td>
<td>uint</td>
<td>integer(4)</td>
<td>target task</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>integer(4)</td>
<td>pad (padding alignment for FORTRAN only)</td>
</tr>
<tr>
<td>hdr_hdl</td>
<td>lapi_long_t</td>
<td>integer(8)</td>
<td>header handler to invoke at target</td>
</tr>
<tr>
<td>uhdr_len</td>
<td>uint</td>
<td>integer(4)</td>
<td>user header length (multiple of processor’s word size)</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>integer(4)</td>
<td>pad2 (padding alignment for 64-bit FORTRAN only)</td>
</tr>
<tr>
<td>uhdr</td>
<td>void *</td>
<td>integer(4) (32-bit)</td>
<td>pointer to user header</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>udata</td>
<td>void *</td>
<td>integer(4) (32-bit)</td>
<td>pointer to user data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>udata_len</td>
<td>ulong</td>
<td>integer(4) (32-bit)</td>
<td>user data length</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>shdlr</td>
<td>scompl_hndlr_t *</td>
<td>integer(4) (32-bit)</td>
<td>send completion handler (optional)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>sinfo</td>
<td>void *</td>
<td>integer(4) (32-bit)</td>
<td>data pointer to pass to send completion handler (optional)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>tgt_cntr</td>
<td>lapi_long_t</td>
<td>integer(8)</td>
<td>target counter (optional)</td>
</tr>
<tr>
<td>org_cntr</td>
<td>lapi_cntr_t *</td>
<td>integer(4) (32-bit)</td>
<td>origin counter (optional)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
</tbody>
</table>
Table 48. The lapi_amx_t fields (continued)

<table>
<thead>
<tr>
<th>lapi_amx_t field name (C)</th>
<th>lapi_amx_t field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>LAPI_Xfer usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpl_cntr</td>
<td>lapi_cntr_t *</td>
<td>integer(4) (32-bit)</td>
<td>completion counter (optional)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>dgsp</td>
<td>lapi_dg_handle_t</td>
<td>integer(4) (32-bit)</td>
<td>handle of a registered DGSP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>status</td>
<td>lapi_status_t</td>
<td>integer(4) (32-bit)</td>
<td>status to return</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>send_offset_dgsp_bytes</td>
<td>ulong</td>
<td>integer(4) (32-bit)</td>
<td>the initial offset bytes that are used to run the DGSP before the data is transferred on the sending side</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
</tbody>
</table>

For additional information, see “Extended user header support” on page 23.

**lapi_get_t details**

Table 49 shows the correspondence among the parameters of the LAPI_Get subroutine, the fields of the C lapi_get_t structure and their datatypes, and the equivalent FORTRAN datatypes. The lapi_get_t fields are listed in the order that they occur in the lapi_xfer_t structure.

Table 49. LAPI_Get and lapi_get_t equivalents

<table>
<thead>
<tr>
<th>lapi_get_t field name (C)</th>
<th>lapi_get_t field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>Equivalent LAPI_Get parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xfer_type</td>
<td>lapi_xfer_type_t</td>
<td>integer(4)</td>
<td>implicit in C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_Xfer value in FORTRAN: LAPI_GET_XFER</td>
</tr>
<tr>
<td>flags</td>
<td>int</td>
<td>integer(4)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_Xfer parameter in FORTRAN: flags</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See “Setting the flags field” on page 273 for more information.</td>
</tr>
<tr>
<td>tgt</td>
<td>uint</td>
<td>integer(4)</td>
<td>tgt</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>integer(4)</td>
<td>LAPI_Xfer parameter in FORTRAN: pad</td>
</tr>
<tr>
<td>tgt_addr</td>
<td>lapi_long_t</td>
<td>integer(8)</td>
<td>tgt_addr</td>
</tr>
<tr>
<td>org_addr</td>
<td>void *</td>
<td>integer(4) (32-bit)</td>
<td>org_addr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>len</td>
<td>ulong</td>
<td>integer(4) (32-bit)</td>
<td>len</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>tgt_cntr</td>
<td>lapi_long_t</td>
<td>integer(8)</td>
<td>tgt_cntr</td>
</tr>
<tr>
<td>org_cntr</td>
<td>lapi_cntr_t *</td>
<td>integer(4) (32-bit)</td>
<td>org_cntr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>chndlr</td>
<td>compl_hndlr_t *</td>
<td>integer(4) (32-bit)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td>LAPI_Xfer parameter in FORTRAN: chndlr</td>
</tr>
</tbody>
</table>
Table 49. LAPI_Get and lapi_get_t equivalents (continued)

<table>
<thead>
<tr>
<th>lapi_get_t field name (C)</th>
<th>lapi_get_t field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>Equivalent LAPI_Get parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>cinfo</td>
<td>void *</td>
<td>integer(4) (32-bit)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td>LAPI_Xfer parameter in FORTRAN: cinfo</td>
</tr>
</tbody>
</table>

When the origin data buffer has completely arrived, the pointer to the completion handler (chndlr) is called with the completion data (cinfo), if chndlr is not a Null pointer (in C) or LAPI_ADDR_NULL (in FORTRAN). Otherwise, the behavior is identical to that of LAPI_Get.

The completion handler that chndlr points to is always executed inline on the origin task.

lapi_getv_t details

Table 50 shows the correspondence among the parameters of the LAPI_Getv subroutine, the fields of the C lapi_getv_t structure and their datatypes, and the equivalent FORTRAN datatypes. The lapi_getv_t fields are listed in Table 49 on page 267 in the order that they occur in the lapi_xfer_t structure.

Table 50. LAPI_Getv and lapi_getv_t equivalents

<table>
<thead>
<tr>
<th>lapi_getv_t field name (C)</th>
<th>lapi_getv_t field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>Equivalent LAPI_Getv parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xfer_type</td>
<td>lapi_xfer_type_t</td>
<td>integer(4)</td>
<td>implicit in C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_Xfer value in FORTRAN:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_GETV_XFER</td>
</tr>
<tr>
<td>flags</td>
<td>int</td>
<td>integer(4)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_Xfer parameter in FORTRAN: flags</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See “Setting the flags field” on page 273 for more information.</td>
</tr>
<tr>
<td>tgt</td>
<td>uint</td>
<td>integer(4)</td>
<td>tgt</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>integer(4)</td>
<td>LAPI_Xfer parameter in FORTRAN: tgt</td>
</tr>
<tr>
<td></td>
<td>lapi_vec_t *</td>
<td>integer(4) (32-bit)</td>
<td>org_vec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>tgt_vec</td>
<td>void *</td>
<td>integer(4) (32-bit)</td>
<td>tgt_vec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>org_cntr</td>
<td>lapi_cntr_t *</td>
<td>integer(4) (32-bit)</td>
<td>org_cntr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>chndlr</td>
<td>compl_hndlr_t *</td>
<td>integer(4) (32-bit)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td>LAPI_Xfer parameter in FORTRAN: chndlr</td>
</tr>
</tbody>
</table>
Table 50. LAPI_Getv and lapi_getv_t equivalents (continued)

<table>
<thead>
<tr>
<th>lapi_getv_t field name (C)</th>
<th>lapi_getv_t field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>Equivalent LAPI_Getv parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>cinfo</td>
<td>void *</td>
<td>integer(4) (32-bit)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td>LAPI_Xfer parameter in FORTRAN: cinfo</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>integer(4)</td>
<td>LAPI_Xfer parameter in FORTRAN (32-bit): pad2</td>
</tr>
</tbody>
</table>

The flags field accepts USE_TGT_VEC_TYPE (see “Setting the flags field” on page 273) to indicate that tgt_vec is to be interpreted as type lapi_vec_t: otherwise, it is interpreted as type lapi_lvec_t. Note the corresponding field is lapi_vec_t in the LAPI_Getv call.

When the origin data buffer has completely arrived, the pointer to the completion handler (chndlr) is called with the completion data (cinfo) if chndlr is not a Null pointer (in C) or LAPI_ADDR_NULL (in FORTRAN). Otherwise, the behavior is identical to that of LAPI_Getv.

The completion handler that chndlr points to is always executed inline on the origin task.

lapi_put_t details

Table 51 shows the correspondence among the parameters of the LAPI_Put subroutine, the fields of the C lapi_put_t structure and their datatypes, and the equivalent FORTRAN datatypes. The lapi_put_t fields are listed in Table 51 in the order that they occur in the lapi_xfer_t structure.

Table 51. LAPI_Put and lapi_put_t equivalents

<table>
<thead>
<tr>
<th>lapi_put_t field name (C)</th>
<th>lapi_put_t field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>Equivalent LAPI_Put parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xfer_type</td>
<td>lapi_xfer_type_t</td>
<td>integer(4)</td>
<td>implicit in C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_Xfer value in FORTRAN: LAPI_PUT_XFER</td>
</tr>
<tr>
<td>flags</td>
<td>int</td>
<td>integer(4)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_Xfer parameter in FORTRAN: flags</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See “Setting the flags field” on page 273 for more information.</td>
</tr>
<tr>
<td>tgt</td>
<td>uint</td>
<td>integer(4)</td>
<td>tgt</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>integer(4)</td>
<td>LAPI_Xfer parameter in FORTRAN: pad</td>
</tr>
<tr>
<td>tgt_addr</td>
<td>lapi_long_t</td>
<td>integer(8)</td>
<td>tgt_addr</td>
</tr>
<tr>
<td>org_addr</td>
<td>void *</td>
<td>integer(4) (32-bit)</td>
<td>org_addr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>len</td>
<td>ulong</td>
<td>integer(4) (32-bit)</td>
<td>len</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>shdlr</td>
<td>scompl_hndlr_t *</td>
<td>integer(4) (32-bit)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td>LAPI_Xfer parameter in FORTRAN: shdlr</td>
</tr>
</tbody>
</table>
### LAPI_Xfer

Table 51. LAPI_Put and lapi_put_t equivalents (continued)

<table>
<thead>
<tr>
<th>lapi_put_t field name (C)</th>
<th>lapi_put_t field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>Equivalent LAPI_Put parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>sinfo</td>
<td>void *</td>
<td>integer(4) (32-bit)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td>LAPI_Xfer parameter in FORTRAN: sinfo</td>
</tr>
<tr>
<td>tgt_cntr</td>
<td>lapi_long_t</td>
<td>integer(8)</td>
<td>tgt_cntr</td>
</tr>
<tr>
<td>org_cntr</td>
<td>lapi_cntr_t *</td>
<td>integer(4) (32-bit)</td>
<td>org_cntr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>cmpl_cntr</td>
<td>lapi_cntr_t *</td>
<td>integer(4) (32-bit)</td>
<td>cmpl_cntr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
</tbody>
</table>

When the origin data buffer is free to be used, the pointer to the send completion handler (shdlr) is called with the send completion data (sinfo), if shdlr is not a Null pointer (in C) or LAPI_ADDR_NULL (in FORTRAN). Otherwise, the behavior is identical to that of LAPI_Put.

### lapi_putv_t details

Table 52 shows the correspondence among the parameters of the LAPI_Putv subroutine, the fields of the C lapi_putv_t structure and their datatypes, and the equivalent FORTRAN datatypes. The lapi_putv_t fields are listed in Table 51 on page 269 in the order that they occur in the lapi_xfer_t structure.

Table 52. LAPI_Putv and lapi_putv_t equivalents

<table>
<thead>
<tr>
<th>lapi_putv_t field name (C)</th>
<th>lapi_putv_t field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>Equivalent LAPI_Putv parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xfer_type</td>
<td>lapi_xfer_type_t</td>
<td>integer(4)</td>
<td>implicit in C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_Xfer value in FORTRAN:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_PUTV_XFER</td>
</tr>
<tr>
<td>flags</td>
<td>int</td>
<td>integer(4)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_Xfer parameter in FORTRAN: flags</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See <a href="#">“Setting the flags field” on page 273</a> for more information.</td>
</tr>
<tr>
<td>tgt</td>
<td>uint</td>
<td>integer(4)</td>
<td>tgt</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>integer(4)</td>
<td>LAPI_Xfer parameter in FORTRAN (64-bit): pad</td>
</tr>
<tr>
<td>shdlr</td>
<td>scompl_hndlr_t *</td>
<td>integer(4) (32-bit)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td>LAPI_Xfer parameter in FORTRAN: shdlr</td>
</tr>
<tr>
<td>sinfo</td>
<td>void *</td>
<td>integer(4) (32-bit)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td>LAPI_Xfer parameter in FORTRAN: sinfo</td>
</tr>
<tr>
<td>org_vec</td>
<td>lapi_vec_t *</td>
<td>integer(4) (32-bit)</td>
<td>org_vec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>tgt_vec</td>
<td>void *</td>
<td>integer(4) (32-bit)</td>
<td>tgt_vec</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>integer(4)</td>
<td>LAPI_Xfer parameter in FORTRAN (32-bit): pad</td>
</tr>
</tbody>
</table>
The flags field accepts USE_TGT_VEC_TYPE (see “Setting the flags field” on page 273) to indicate that tgt_vec is to be interpreted as lapi_vec_t; otherwise, it is interpreted as a lapi_vec_t. Note that the corresponding field is lapi_vec_t in the LAPI_Putv call.

When the origin data buffer is free to be modified, the pointer to the send completion handler (shdlr) is called with the send completion data (sinfo), if shdlr is not a Null pointer (in C) or LAPI_ADDR_NULL (in FORTRAN). Otherwise, the behavior is identical to that of LAPI_Putv.

lapi_hwxfer_t details
This structure applies to HPS systems running RSCT LAPI for AIX.

RSCT LAPI for AIX allows user-initiated RDMA “get” and “put” transfers. RDMA transfer is initiated with LAPI_Xfer using the LAPI_RDMA_XFER transfer type and the lapi_hwxfer_t structure. The interface for the RMDA transfer operation is similar to the “get” and “put” transfer operations with the following differences. The operation type is specified by setting the op field of lapi_hwxfer_t to either LAPI_RDMA_GET or LAPI_RDMA_PUT. Optionally, the user can logically OR LAPI_RCNTR_UPDATE, LAPI_RDCALLBACK, or both with this op field to specify a target notification request. When target notification is requested, the callback (or counter) registered with the tag value on the target side is called (or incremented). Also, the source and target buffers for the LAPI_RDMA_GET and LAPI_RDMA_PUT operations are specified with PVO/offset combinations rather than virtual addresses. Finally, because the RDMA transport requires remote adapter resources, a remote rCxt must be specified. The tag registration, PVO, and remote rCxt are obtained using RDMA-related LAPI_Util functions.

LAPI_Xfer with transfer type LAPI_RDMA_XFER is not supported for persistent LAPI jobs.

Table 53 shows the correspondence among the parameters of the LAPI_Xfer subroutine with transfer type LAPI_RDMA_XFER, the fields of the C lapi_hwxfer_t structure and their datatypes, and the equivalent FORTRAN datatypes. The lapi_hwxfer_t fields are listed in Table 53 in the order that they occur in the lapi_xfer_t structure.

Table 53. The lapi_hwxfer_t fields

<table>
<thead>
<tr>
<th>lapi_hwxfer_t field name (C)</th>
<th>lapi_hwxfer_t field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>LAPI_Xfer usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xfer_type</td>
<td>lapi_xfer_type_t</td>
<td>integer(4)</td>
<td>LAPI_RDMA_XFER</td>
</tr>
<tr>
<td>tgt</td>
<td>uint</td>
<td>integer(4)</td>
<td>The target task</td>
</tr>
<tr>
<td>op</td>
<td>lapi_rdma_op_t</td>
<td>integer(4)</td>
<td>The type of operation</td>
</tr>
<tr>
<td>rdma_tag</td>
<td>lapi_rdma_tag_t</td>
<td>integer(4)</td>
<td>The user RDMA tag</td>
</tr>
</tbody>
</table>
**LAPI_Xfer**

Table 53. The lapi_hwxfer_t fields (continued)

<table>
<thead>
<tr>
<th>lapi_hwxfer_t field name (C)</th>
<th>lapi_hwxfer_t field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>LAPI_Xfer usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>remote_cxt</td>
<td>lapi_user_cxt_t</td>
<td>integer(4)</td>
<td>The context on the remote side</td>
</tr>
<tr>
<td>src_pvo</td>
<td>lapi_user_pvo_t</td>
<td>integer(8)</td>
<td>The initial PVO</td>
</tr>
<tr>
<td>tgt_pvo</td>
<td>lapi_user_pvo_t</td>
<td>integer(8)</td>
<td>The target PVO</td>
</tr>
<tr>
<td>src_offset</td>
<td>uint</td>
<td>integer(4)</td>
<td>The source offset</td>
</tr>
<tr>
<td>tgt_offset</td>
<td>uint</td>
<td>integer(4)</td>
<td>The target offset</td>
</tr>
<tr>
<td>len</td>
<td>uint</td>
<td>integer(4)</td>
<td>The length of the data to be sent</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>integer(4)</td>
<td>pad (padding alignment for FORTRAN only)</td>
</tr>
<tr>
<td>org_cntr</td>
<td>lapi_cntr_t *</td>
<td>integer(4) (32-bit)</td>
<td>The origin counter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>shdlr</td>
<td>scompl_hdlr_t *</td>
<td>integer(4) (32-bit)</td>
<td>The send completion handler</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>info</td>
<td>void *</td>
<td>integer(4) (32-bit)</td>
<td>The send completion data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
</tbody>
</table>

**lapi_rmw_t details**

Table 54 shows the correspondence among the parameters of the LAPI_Rmw subroutine, the fields of the C lapi_rmw_t structure and their datatypes, and the equivalent FORTRAN datatypes. The lapi_rmw_t fields are listed in Table 54 in the order that they occur in the lapi_xfer_t structure.

Table 54. LAPI_Rmw and lapi_rmw_t equivalents

<table>
<thead>
<tr>
<th>lapi_rmw_t field name (C)</th>
<th>lapi_rmw_t field type (C)</th>
<th>Equivalent FORTRAN datatype</th>
<th>Equivalent LAPI_Rmw parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xfer_type</td>
<td>lapi_xfer_type_t</td>
<td>integer(4)</td>
<td>implicit in C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_Xfer value in FORTRAN: LAPI_RMW_XFER</td>
</tr>
<tr>
<td>op</td>
<td>RMW_ops_t</td>
<td>integer(4)</td>
<td>op</td>
</tr>
<tr>
<td>tgt</td>
<td>uint</td>
<td>integer(4)</td>
<td>tgt</td>
</tr>
<tr>
<td>size</td>
<td>uint</td>
<td>integer(4)</td>
<td>implicit in C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LAPI_Xfer parameter in FORTRAN: size (must be 32 or 64)</td>
</tr>
<tr>
<td>tgt_var</td>
<td>lapi_long_t</td>
<td>integer(8)</td>
<td>tgt_var</td>
</tr>
<tr>
<td>in_val</td>
<td>void *</td>
<td>integer(4) (32-bit)</td>
<td>in_val</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>prev_tgt_val</td>
<td>void *</td>
<td>integer(4) (32-bit)</td>
<td>prev_tgt_val</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>org_cntr</td>
<td>lapi_cntr_t *</td>
<td>integer(4) (32-bit)</td>
<td>org_cntr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td></td>
</tr>
<tr>
<td>shdlr</td>
<td>scompl_hdlr_t *</td>
<td>integer(4) (32-bit)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer(8) (64-bit)</td>
<td>LAPI_Xfer parameter in FORTRAN: shdlr</td>
</tr>
</tbody>
</table>
When the origin data buffer is free to be used, the pointer to the send completion handler (shdlr) is called with the send completion data (sinfo), if shdlr is not a Null pointer (in C) or LAPI_ADDR_NULL (in FORTRAN). The size value must be either 32 or 64, indicating whether you want the in_val and prev_tgt_val fields to point to a 32-bit or 64-bit quantity, respectively. Otherwise, the behavior is identical to that of LAPI_Rmw.

### Setting the flags field

You can set one or more flags using the | (bitwise or) operator. User directives are always followed and could result in incorrect results if used improperly. Appropriate hints might improve performance, but they may be ignored by LAPI. Inappropriate hints might degrade performance, but they will not cause incorrect results.

The following directive flags are defined:

- **USE_TGT_VEC_TYPE**
  Instructs LAPI to use the vector type of the target vector (tgt_vec). In other words, tgt_vec is to be interpreted as type lapi_vec_t; otherwise, it is interpreted as type lapi_lvec_t. The lapi_lvec_t type uses lapi_long_t. The lapi_vec_t type uses void * or long. Incorrect results will occur if one type is used in place of the other.

- **BUFFER_BOTH_CONTIGUOUS**
  Instructs LAPI to treat all data to be transferred as contiguous, which can improve performance. If this flag is set when non-contiguous data is sent, data will likely be corrupted.

The following hint flags apply to HPS systems running RSCT LAPI for AIX:

- **LAPI_NOT_USE_BULK_XFER**
  Instructs LAPI not to use bulk transfer, independent of the current setting for the task.

- **LAPI_USE_BULK_XFER**
  Instructs LAPI to use bulk transfer, independent of the current setting for the task.

If neither of these hint flags is set, LAPI will use the behavior defined for the task. If both of these hint flags are set, LAPI_NOT_USE_BULK_XFER will take precedence.

These hints may or may not be honored by the communication library.

### lapi_mc_t details

With multicasting, one sender from a group sends a message to all of the group members including itself.
Use this transfer type to transfer data to a group of target tasks where a handler runs at the target task when the message arrives at the target task and where an optional completion handler can run after the message delivery completes. **LAPI_MC_XFER** allows the user to provide a header handler and optional completion handler. The header handler is used to specify the target buffer address for writing the data, eliminating the need to know the address on the origin task when the subroutine is called.

**Table 55** shows the correspondence among the parameters of the **LAPI_Xfer** subroutine with transfer type **LAPI_MC_XFER**, the fields of the C **lapi_mc_t** structure and their datatypes, and the equivalent FORTRAN datatypes. The **lapi_mc_t** fields are listed in **Table 55** in the order that they occur in the **lapi_xfer_t** structure.

<table>
<thead>
<tr>
<th><strong>lapi_mc_t field name</strong> (C)</th>
<th><strong>lapi_mc_t field type</strong> (C)</th>
<th><strong>Equivalent FORTRAN datatype</strong></th>
<th><strong>LAPI_Xfer usage</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Xfer_type</strong></td>
<td>lapi_xfer_type_t</td>
<td>integer(4)</td>
<td>LAPI_MC_XFER (MC)</td>
</tr>
<tr>
<td><strong>flags</strong></td>
<td>int</td>
<td>integer(4)</td>
<td>Indicates whether to use zero copy, for example. See &quot;Setting the flags field&quot; on page 273 for more information.</td>
</tr>
<tr>
<td><strong>group</strong></td>
<td>lapi_group_t</td>
<td>integer(4) (32-bit)</td>
<td>target group</td>
</tr>
<tr>
<td><strong>none</strong></td>
<td>none</td>
<td>integer(4) (64-bit)</td>
<td><strong>LAPI_Xfer</strong> parameter in FORTRAN (64-bit): pad</td>
</tr>
<tr>
<td><strong>hdr_hdl</strong></td>
<td>lapi_long_t</td>
<td>integer(8)</td>
<td>active message header handler</td>
</tr>
<tr>
<td><strong>uhdr_len</strong></td>
<td>uint</td>
<td>integer(4)</td>
<td>user header length (multiple of processor’s word size)</td>
</tr>
<tr>
<td><strong>uhdr</strong></td>
<td>void *</td>
<td>integer(4) (32-bit)</td>
<td>user header data</td>
</tr>
<tr>
<td><strong>userdata</strong></td>
<td>void *</td>
<td>integer(4) (64-bit)</td>
<td>user data to be transferred</td>
</tr>
<tr>
<td><strong>udata_len</strong></td>
<td>ulong</td>
<td>integer(4) (32-bit)</td>
<td>transfer length</td>
</tr>
<tr>
<td><strong>shdlr</strong></td>
<td>scompl_hndlr_t *</td>
<td>integer(4) (32-bit)</td>
<td>send completion handler</td>
</tr>
<tr>
<td><strong>sinfo</strong></td>
<td>void *</td>
<td>integer(4) (64-bit)</td>
<td>send completion data</td>
</tr>
</tbody>
</table>

**Restrictions**  
**LAPI_Xfer** with transfer type **LAPI_RDMA_XFER** is not supported for persistent LAPI jobs.

**Return values**

**LAPI_SUCCESS**  
Indicates that the function call completed successfully.

**LAPI_ERR_DATA_LEN**  
Indicates that the value of **udata_len** or **len** is greater than the value of LAPI constant **LAPI_MAX_MSG_SZ**.
<table>
<thead>
<tr>
<th>LAPI_ERR_DGSP</th>
<th>Indicates that the DGSP that was passed in is Null (in C) or LAPI_ADDR_NULL (in FORTRAN) or is not a registered DGSP.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPI_ERR_DGSP_ATOM</td>
<td>Indicates that the DGSP has an atom_size that is less than 0 or greater than MAX_ATOM_SIZE.</td>
</tr>
<tr>
<td>LAPI_ERR_DGSP_BRANCH</td>
<td>Indicates that the DGSP attempted a branch that fell outside the code array.</td>
</tr>
<tr>
<td>LAPI_ERR_DGSP_CTL</td>
<td>Indicates that a DGSP control instruction was encountered in a non-valid context (such as a gather-side control or scatter-side control with an atom size of 0 at gather, for example).</td>
</tr>
<tr>
<td>LAPI_ERR_DGSP_OPC</td>
<td>Indicates that the DGSP op-code is not valid.</td>
</tr>
<tr>
<td>LAPI_ERR_DGSP_STACK</td>
<td>Indicates that the DGSP has greater GOSUB depth than the allocated stack supports. Stack allocation is specified by the dgsp-&gt;depth member.</td>
</tr>
<tr>
<td>LAPI_ERR_GRP</td>
<td>Indicates that the group passed in is not valid (not initialized or already deallocated).</td>
</tr>
<tr>
<td>LAPI_ERR_HDR_HNDLR_NULL</td>
<td>Indicates that the hdr_hdl passed in is Null (in C) or LAPI_ADDR_NULL (in FORTRAN).</td>
</tr>
<tr>
<td>LAPI_ERR_HNDL_INVALID</td>
<td>Indicates that the hndl passed in is not valid (not initialized or in terminated state).</td>
</tr>
<tr>
<td>LAPI_ERR_IN_VAL_NULL</td>
<td>Indicates that the in_val pointer is Null (in C) or LAPI_ADDR_NULL (in FORTRAN).</td>
</tr>
<tr>
<td>LAPI_ERR_LW_DATA_LEN</td>
<td>An incorrect data length, header length, or both were specified for LAPI_AM_LW_XFER.</td>
</tr>
<tr>
<td>LAPI_ERR_LW_NO_HNDLR_SET</td>
<td>No header handler has been set for using LAPI_AM_LW_XFER.</td>
</tr>
<tr>
<td>LAPI_ERR_MEMORY_EXHAUSTED</td>
<td>LAPI is unable to obtain memory from the system.</td>
</tr>
<tr>
<td>LAPI_ERR_OFFSET_LEN</td>
<td>Indicates that the DGSP send offset is out of range (applies to LAPI_AMX_XFER only).</td>
</tr>
<tr>
<td>LAPI_ERR_OP_SZ</td>
<td>Indicates that the lapi_rmw_t size field is not set to 32 or 64.</td>
</tr>
<tr>
<td>LAPI_ERR_ORG_ADDR_NULL</td>
<td>Indicates either that the udata parameter passed in is Null (in C) or LAPI_ADDR_NULL (in FORTRAN) and udata_len is greater than 0, or that the org_addr passed in is Null (in C) or LAPI_ADDR_NULL (in FORTRAN) and len is greater than 0.</td>
</tr>
<tr>
<td>Note: if Xfer_type = LAPI_DGSP_XFER, the case in which udata is Null (in C) or LAPI_ADDR_NULL (in FORTRAN) and udata_len is greater than 0 is valid, so an error is not returned.</td>
<td></td>
</tr>
</tbody>
</table>
| LAPI_ERR_ORG_EXTENT    | Indicates that the org_vec's extent (stride *
LAPI_Xfer

num_vecs) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_ORG_STRIDE  Indicates that the org_vec stride is less than block.
LAPI_ERR_ORG_VEC_ADDR Indicates that the org_vec->info[i] is Null (in C) or LAPI_ADDR_NULL (in FORTRAN), but its length (org_vec->len[i]) is not 0.
LAPI_ERR_ORG_VEC_LEN Indicates that the sum of org_vec->len is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.
LAPI_ERR_ORG_VEC_NULL Indicates that the org_vec value is Null (in C) or LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_ORG_VEC_TYPE Indicates that the org_vec->vec_type is not valid.
LAPI_ERR_RMW_OP Indicates that the op is not valid.
LAPI_ERR_STRIDE_ORG_VEC_ADDR_NULL Indicates that the strided vector address org_vec->info[0] is Null (in C) or LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_STRIDE_TGT_VEC_ADDR_NULL Indicates that the strided vector address tgt_vec->info[0] is Null (in C) or LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_TGT Indicates that the tgt passed in is outside the range of tasks defined in the job.
LAPI_ERR_TGT_ADDR_NULL Indicates that ret_addr is Null (in C) or LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_TGT_EXTENT Indicates that tgt_vec's extent (stride * num_vecs) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.
LAPI_ERR_TGT_PURGED Indicates that the subroutine returned early because LAPI_Purge_totask() was called.
LAPI_ERR_TGT_STRIDE Indicates that the tgt_vec stride is less than block.
LAPI_ERR_TGT_VAR_NULL Indicates that the tgt_var address is Null (in C) or that the value of tgt_var is LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_TGT_VEC_ADDR Indicates that the tgt_vec->info[i] is Null (in C) or LAPI_ADDR_NULL (in FORTRAN), but its length (tgt_vec->len[i]) is not 0.
LAPI_ERR_TGT_VEC_LEN Indicates that the sum of tgt_vec->len is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.
LAPI_ERR_TGT_VEC_NULL Indicates that tgt_vec is Null (in C) or LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_TGT_VEC_TYPE Indicates that the tgt_vec->vec_type is not valid.
LAPI_ERR_UHDR_LEN Indicates that the uhdr_len value passed in is
greater than MAX_UHDR_SZ or is not a multiple of the processor’s word size.

**LAPI_ERR_UHDR_NULL** Indicates that the _uhdr_ passed in is Null (in C) or **LAPI_ADDR_NULL** (in FORTRAN), but _uhdr_len_ is not 0.

**LAPI_ERR_VEC_LEN_DIFF** Indicates that _org_vec_ and _tgt_vec_ have different lengths (_len[]_).

**LAPI_ERR_VEC_NUM_DIFF** Indicates that _org_vec_ and _tgt_vec_ have different _num_secs_.

**LAPI_ERR_VEC_TYPE_DIFF** Indicates that _org_vec_ and _tgt_vec_ have different vector types (_vec_type_).

**LAPI_ERR_XFER_CMD** Indicates that the _Xfer_cmd_ is not valid.

**Location**

/usr/lib/liblapi_r.a

**C examples**

1. To run the sample code shown in **LAPI_Get** using the **LAPI_Xfer** interface:

   ```c
   { 
      lapi_xfer_t xfer_struct;
      /* initialize the table buffer for the data addresses */
      /* get remote data buffer addresses */
      LAPI_Address_init(hndl,(void *)data_buffer,data_buffer_list);
      /* retrieve _data_len_ bytes from address _data_buffer_list_[tgt]_ on */
      /* task _tgt_. write the data starting at address _data_buffer_. */
      /* _tgt_cntr_ and _org_cntr_ can be Null. */
      xfer_struct.Get.Xfer_type = LAPI_GET_XFER;
      xfer_struct.Get.flags = 0;
      xfer_struct.Get.tgt = tgt;
      xfer_struct.Get.tgt_addr = data_buffer_list[tgt];
      xfer_struct.Get.org_addr = data_buffer;
      xfer_struct.Get.len = data_len;
      xfer_struct.Get.tgt_cntr = tgt_cntr;
      xfer_struct.Get.org_cntr = org_cntr;
      LAPI_Xfer(hndl, &xfer_struct);
   }
   ```

2. To implement the **LAPI_STRIDED_VECTOR** example from **LAPI_Amsendv** using the **LAPI_Xfer** interface:

   ```c
   { 
      lapi_xfer_t xfer_struct; /* info for LAPI_xfer call */
      lapi_vec_t vec; /* data for data transfer */
      vec->num_vecs = NUM_VECS; /* NUM_VECS = number of vectors to transfer */
      /* must match that of the target vector */
   }
   ```
vec->vec_type = LAPI_GEN_STRIDED_XFER;       /* same as target vector */

vec->info[0] = buffer_address; /* starting address for data copy */
vec->info[1] = block_size; /* bytes of data to copy */
vec->info[2] = stride; /* distance from copy block to copy block */
/* data will be copied as follows: */
/* block_size bytes will be copied from buffer_address */
/* block_size bytes will be copied from buffer_address+stride */
/* block_size bytes will be copied from buffer_address+(2*stride) */
/* block_size bytes will be copied from buffer_address+(3*stride) */
/* block_size bytes will be copied from buffer_address+((NUM_VECS-1)*stride) */
/* block_size bytes will be copied from buffer_address+((NUM_VECS-1)*stride) */
xfer_struct.Amv.Xfer_type = LAPI_AMV_XFER;
xfer_struct.Amv.flags = 0;
xfer_struct.Amv.tgt = tgt;
xfer_struct.Amv.hdr_hdl = hdr_hdl_list[tgt];
xfer_struct.Amv.uhdr_len = uhdr_len; /* user header length */
xfer_struct.Amv.uhdr = uhdr;

/* LAPI_AMV_XFER allows the use of a send completion handler */
/* If non-Null, the shdlr function is invoked at the point */
/* the origin counter would increment. Note that both the */
/* org_cntr and shdlr can be used. */
/* The user's shdlr must be of type scompl_hndlr_t *. */
/* scompl_hndlr_t is defined in /usr/include/lapi.h */
xfer_struct.shdlr = shdlr;

/* Use sinfo to pass user-defined data into the send */
/* completion handler, if desired. */
xfer_struct.sinfo = sinfo; /* send completion data */

xfer_struct.org_vec = vec;
xfer_struct.tgt_cntr = tgt_cntr;
xfer_struct.org_cntr = org_cntr;
xfer_struct.cmpl_cntr = cmpl_cntr;

LAPI_Xfer(hndl, &xfer_struct);

See LAPI_Amsendv for more information about the header handler definition.

3. To transfer data to a group of target tasks:
   #define MC_HDLR 1
   ...
   /* header handler routine to execute on target task */
   void *hdr_hndlr(lapi_handle_t *hndl, void *uhdr, uint *uhdr_len, ulong *msg_len,
                   compl_hndlr_t **cmpl_hndlr, void **user_info)
   {
     /* set completion handler pointer and other information */
     /* return base address for LAPI to begin its data copy */
   }
   {
     lapi_handle_t hndl; /* the LAPI handle */
```c
int task_id; /* the LAPI task ID */
int num_tasks; /* the total number of tasks */
int i;
void *hdr_hdlr_list[NUM_TASKS]; /* the table of remote header handlers */
lapi_group_t group; /* the group handle */
int task_list[NUM_TASKS]; /* array to store task ID */
int data_buffer[DATA_LEN]; /* the data to transfer */

/* retrieve header handler addresses */
LAPI_Addr_set(hndl, (void*)hdr_hdlr, MC_HDLR);
for (i=0; i<NUM_TASKS; i++)
    task_list[i] = i;
LAPI_Group_create(hndl, NUM_TASKS, task_list, &group);
/*
** up to this point, all instructions have executed on all
** tasks. we now begin differentiating tasks.
*/
if (sender) { /* origin task */
    /* initialize data buffer, etc. */
    
    lapi_xfer_t xfer_struct;
xfer_struct.MC.Xfer_type = LAPI_MC_XFER;
xfer_struct.MC.flags = 0;
xfer_struct.MC.group = group;
xfer_struct.MC.hdr_hdl = MC_HDLR;
xfer_struct.MC.uhdr = Null;
xfer_struct.MC.uhdr_len = 0;
xfer_struct.MC.udata = data_buffer;
xfer_struct.MC.udata_len = DATA_LEN*sizeof(int));
xfer_struct.MC.udata_len = DATA_LEN*sizeof(int));
xfer_struct.MC.shdlr = Null;
xfer_struct.MC.sinfo = Null;
LAPI_Xfer(hndl, &xfer_struct);
} else { /* receivers */
    
    
    LAPI_Group_free(hndl, group)
    
    

Related information

"Bulk message transfer on AIX" on page 63

Subroutines: LAPI_Amsend, LAPI_Amsendv, LAPI_Get, LAPI_Getv,
LAPI_Group_create, LAPI_Group_free, LAPI_Put, LAPI_Putv, LAPI_Rmw
```
Chapter 21. Subroutines for standalone systems

Use the subroutines in this chapter on standalone systems.
LAPI_Nopoll_wait

Purpose

Waits for a counter update without polling.

Library

Availability Library (liblapi_r.a)

C syntax

```c
#include <lapi.h>

void LAPI_Nopoll_wait(
    hndl,  // lapi_handle_t
    cntr_ptr,  // lapi_cntr_t *
    val,  // int
    cur_cntr_val  // int *
)
```

FORTRAN syntax

```fortran
include 'lapif.h'

lapi_nopoll_wait(hndl, cntr, val, cur_cntr_val, ierror)
```

Parameters

**Input**

- **hndl** Specifies the LAPI handle.
- **val** Specifies the relative counter value (starting from 1) that the counter needs to reach before returning.
- **cur_cntr_val** Specifies the integer value of the current counter. The value of this parameter can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

**Input/output**

- **cntr_ptr** Points to the lapi_cntr_t structure in C.
- **cntr** Is the lapi_cntr_t structure in FORTRAN.

**Output**

- **ierror** Specifies a FORTRAN return code. This is always the last parameter.

Description

**Type of call:** recovery (blocking)

This subroutine waits for a counter update without polling (that is, without explicitly invoking LAPI's internal communication dispatcher). This call may or may not check for message arrivals over the LAPI context hndl. The cur_cntr_val variable is set to the current counter value. Although it has higher latency than LAPI_Waitcnt, LAPI_Nopoll_wait frees up the processor for other uses.
Note: To use this subroutine, the *lib_vers* field in the *lapi_info_t* structure must be set to *L2_LIB* or *LAST_LIB*.

**Restrictions**

Use of this subroutine is *not* recommended on a system that is running Parallel Environment (PE).

**Return values**

- **LAPI_SUCCESS**
  Indicates that the function call completed successfully.

- **LAPI_ERR_CNTR_NULL**
  Indicates that the *cntr_ptr* pointer is NULL (in C) or that the value of *cntr* is *LAPI_ADDR_NULL* (in FORTRAN).

- **LAPI_ERR_CNTR_VAL**
  Indicates that the *val* passed in is less than or equal to 0.

- **LAPI_ERR_HNDL_INVALID**
  Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

- **LAPI_ERR_MULTIPLE_WAITERS**
  Indicates that more than one thread is waiting for the counter.

- **LAPI_ERR_TGT_PURGED**
  Indicates that the subroutine returned early because *LAPI_Purge_totask()* was called.

**Location**

/usr/lib/liblapi_r.a

**Related information**

Subroutines: *LAPI_Init, LAPI_Purge_totask, LAPI_Resume_totask, LAPI_Setcntnrt_wstatus*
**LAPI_Purge_totask**

**Purpose**
Allows a task to cancel messages to a given destination.

**Library**
Availability Library (liblapi_r.a)

**C syntax**
```c
#include <lapi.h>
int LAPI_Purge_totask(hndl, dest)
lapi_handle_t hndl;
uint dest;
```

**FORTRAN syntax**
```fortran
#include 'lapif.h'

lapi_purge_totask(hndl, dest, ierror)
integer hndl
integer dest
integer ierror
```

**Parameters**

**Input**
- `hndl` Specifies the LAPI handle.
- `dest` Specifies the destination instance ID to which pending messages need to be cancelled.

**Output**
- `ierror` Specifies a FORTRAN return code. This is always the last parameter.

**Description**

**Type of call: recovery**

This subroutine cancels messages and resets the state corresponding to messages in flight or submitted to be sent to a particular target task. This is an entirely local operation. For correct behavior a similar invocation is expected on the destination (if it exists). This function cleans up all the state associated with pending messages to the indicated target task. It is assumed that before the indicated task starts communicating with this task again, it also purges this instance (or that it was terminated and initialized again). It will also wake up all threads that are in `LAPI_Nopoll_wait` depending on how the arguments are passed to the `LAPI_Nopoll_wait` function. The behavior of `LAPI_Purge_totask` is undefined if LAPI collective functions are used.

`LAPI_Purge_totask` is normally used after connectivity has been lost between two tasks. If connectivity is restored, the tasks can restored for LAPI communication by calling `LAPI_Resume_totask`.
**LAPI_Purge_totask**

**Restrictions**
This subroutine should not be used when the parallel application is running in an environment that includes PE and TWS LoadLeveler.

**Return values**
- **LAPI_SUCCESS**: Indicates that the function call completed successfully.
- **LAPI_ERR_HNDL_INVALID**: Indicates that the `hndl` passed in is not valid (not initialized or in terminated state).
- **LAPI_ERR_TGT**: Indicates that `dest` is outside the range of tasks defined in the job.

**Location**
/usr/lib/liblapi_r.a

**Related information**
Subroutines: LAPI_Init, LAPI_Nopoll_wait, LAPI_Resume_totask, LAPI_Term
LAPI_Resume_totask

Purpose
Re-enables the sending of messages to the task.

Library
Availability Library (liblapi_r.a)

C syntax
#include <lapi.h>

int LAPI_Resume_totask
    (hndl, dest)

lapi_handle_t hndl;
uint dest;

FORTRAN syntax

include 'lapif.h'

lapi_resume_totask(hndl, dest, ierror)

integer hndl
integer dest
integer ierror

Parameters

Input
hndl Specifies the LAPI handle.
dest Specifies the destination instance ID with which to resume communication.

Output
ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: recovery

This subroutine is used in conjunction with LAPI_Purge_totask. It enables LAPI communication to be reestablished for a task that had previously been purged. The purged task must either restart LAPI or execute a LAPI_Purge_totask/LAPI_Resume_totask sequence for this task.

Restrictions

Use of this subroutine is not recommended on a system that is running Parallel Environment (PE).

Return values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in terminated state).
**LAPI.Resume_totask**

**LAPI_ERR_TGT** Indicates that the tgt passed in is outside the range of tasks defined in the job.

**Location**

/usr/lib/liblapi_r.a

**Related information**

Subroutines: LAPI_Init, LAPI_Nopoll_wait, LAPI_Purge_totask, LAPI_Term
LAPI_Setcntr_wstatus

LAPI_Setcntr_wstatus

Purpose

Used to set a counter to a specified value and to set the associated destination list array and destination status array to the counter.

Library

Availability Library (liblapi_r.a)

C syntax

```
#include <lapi.h>

int LAPI_Setcntr_wstatus(hndl, cntr, num_dest, dest_list, dest_status)
```

FORTRAN syntax

```
include 'lapif.h'

lapi_setcntr_wstatus(hndl, cntr, num_dest, dest_list, dest_status, ierror)
```

Parameters

Input

- `hndl`: Specifies the LAPI handle.
- `num_dest`: Specifies the number of tasks in the destination list.
- `dest_list`: Specifies an array of destinations waiting for this counter update. If the value of this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), no status is returned to the user.

Input/output

- `cntr`: Specifies the address of the counter to be set (in C) or the counter structure (in FORTRAN). The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

Output

- `dest_status`: Specifies an array of status that corresponds to `dest_list`. The value of this parameter can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
- `ierror`: Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: recovery
This subroutine sets `cntr` to 0. Use `LAPI_Setcntr_wstatus` to set the associated destination list array (`dest_list`) and destination status array (`dest_status`) to the counter. Use a corresponding `LAPI_Nopoll_wait` call to access these arrays. These arrays record the status of a task from where the thread calling `LAPI_Nopoll_wait()` is waiting for a response.

The return values for `dest_status` are:

- **LAPI_MSG_INITIAL**: The task is purged or is not received.
- **LAPI_MSG_RECVD**: The task is received.
- **LAPI_MSG_PURGED**: The task is purged, but not received.
- **LAPI_MSG_PURGED_RCVD**: The task is received and then purged.
- **LAPI_MSG_INVALID**: Not valid; the task is already purged.

**Note:** To use this subroutine, the `lib_vers` field in the `lapi_info_t` structure must be set to `L2_LIB` or `LAST_LIB`.

### Restrictions

Use of this subroutine is *not* recommended on a system that is running Parallel Environment (PE).

### Return values

- **LAPI_SUCCESS**: Indicates that the function call completed successfully.
- **LAPI_ERR_CNTR_NULL**: Indicates that the `cntr` value passed in is NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN).
- **LAPI_ERR_HNDL_INVALID**: Indicates that the `hndl` passed in is not valid (not initialized or in terminated state).
- **LAPI_ERR_RET_PTR_NULL**: Indicates that the value of `dest_status` is NULL in C (or `LAPI_ADDR_NULL` in FORTRAN), but the value of `dest_list` is not NULL in C (or `LAPI_ADDR_NULL` in FORTRAN).

### Location

`/usr/lib/liblapi_r.a`

### Related information

Subroutines: `LAPI_Getcntr`, `LAPI_Nopoll_wait`, `LAPI_Purge_totask`, `LAPI_Setcntr`
LAPI_Setcntr_wstatus
Chapter 22. Sample LAPI programs

The sample programs that are shipped with LAPI illustrate LAPI's building blocks. You can use these files to help you with the LAPI programs you create to solve more complex problems. These samples are structured to provide you with a detailed look at basic LAPI operations.

The sample source code resides in several subdirectories. Each subdirectory illustrates some aspect of LAPI, forming an example group. Each example group is a self-contained unit, with a separate README file and Makefile. Because the samples are installed into a common system directory, you should copy the sample tree into your personal file space for your own use. That way, the source files can be changed as needed and then built.

There are two types of example groups:

- Normal LAPI operations
  Showing Initialization and setup (init group), LAPI API communication calls (lapi_api), an example of two-way LAPI communication (basic), illustration of LAPI's DGSP interface (dgsp), illustration of LAPI's vector interface (vector) and illustration of LAPI's xfer interface (xfer)
- Special cases
  Showing operation of:
  - RSCT LAPI for AIX or LAPI for Linux without PE and TWS LoadLeveler (standalone)
  - LAPI support for mixed 32-bit and 64-bit jobs (interop)

The structure of each normal example group is the same. Each directory contains a README, Makefile, and one or more source files. Complete directions for building and executing the examples can be found in the README file in each directory. In each case, building simply requires execution of the make command. Several source files are created.

Sample program directory structure

The sample files install into system directory /opt/rsct/lapi/samples. It is assumed that a non-root user wishing to work with the samples has copied them into their own file space. Throughout the remainder of this discussion, references to samples directories are assumed to be relative to the top directory (the directory from which the user is working). So for example, if the user has copied the contents of /opt/rsct/lapi/samples into directory /u/fred/samples, the use of "init/init.c" in the discussion will refer to init.c that resides in the init subdirectory of /u/fred/samples.

Details of the subdirectories are given here.

init
Contains examples of initialization, setup and termination functions for LAPI.
- The Init.c sample illustrates basic initialization and termination. It also demonstrates support for multiple initialization and termination of LAPI handles. At the first initialization, a user error handler is registered.
- Addr.c demonstrates LAPI's address manipulation functions, including LAPI_Address_Init, LAPI_Address_init64, LAPI_Addr_set and LAPI_Addr_get. Each task declares two local variables then does a collective
exchange of their addresses as a void * (using LAPI_Address_init) and as a lapi_long_t (using LAPI_Address_init64).

Each task then does a LAPI_Addr_set followed by a LAPI_Addr_get to demonstrate the use of indexed address tables.

- Qenv_senv.c illustrates the use of LAPI's runtime query facility. All parameters that can be queried from a LAPI instance are shown in the source code, as well as those that can be set. For each parameter, the default value is printed. For settable parameters, the value is changed and the new value printed to verify the change.

lapi_api

Contains an example demonstrating each LAPI communication API call. Both a C file and a FORTRAN file are included for each API call.

In each example, a buddy system is used for pairing tasks. Each buddy pair has one task that drives communication with the other task in the pair (his buddy). A single communication is made between the driving task and his buddy.

In all cases, a basic setup is done, followed by the API call. Finally, normal LAPI cleanup operations are done. Synchronization fences are used throughout the code.

Each source code file in the lapi_api directory is named for the demonstrated API call. The following source files are provided:
- Am.c, Amf.F - LAPI_Amsend
- Amv.c, Amvf.F - LAPI_Amsendv
- Get.c, Getf.F - LAPI_Get
- Getv.c, Getvf.F - LAPI_Getv
- Put.c, Putf.F - LAPI_Put
- Putv.c, Putvf.F - LAPI_Putv
- Rmw.c, Rmwf.F - LAPI_Rmw
- Rmw64.c, Rmw64f.F - LAPI_Rmw64

For purposes of deeper illustration, the source code of the Am.c is included below with line numbers and complete annotation. The Am samples demonstrate user-defined header and completion handlers. See the description of the basic/accumulate_and_return.Am sample as well as the complete annotated description below for more details on execution sequence of header and completion handlers.

basic

Shows examples of different approaches to the same communication using different LAPI API calls. The examples illustrate two-way communication. The driver sends an array of ints to its buddy. Its buddy accumulates the data with some local data, then sends the data back to the original sender. Different approaches to synchronization are illustrated, depending on the nature of the API call being used. Each source file is named accumulate_and_return.API.c, where API is one of Am, Put or Xfer.

There are three examples of this communication, one using each of the API calls implied by the name:

accumulate_and_return.Am.c

accumulate_and_return.Am.c does a LAPI_Amsend call for the original data transfer. This example also shows how to use user header data as part of the transfer, as well as the use of a completion handler parameter to pass data between the header and completion handlers. Note that user header data will be available in the first packet, and is thus available to the header
Using the combined facility of a user header and a completion handler parameter ensures the delivery of data from the message sender all the way through to the completion handler on the receiver.

This example defines a data type to use for the completion handler parameter. The parameter is passed as a void *. Then it is cast to the newly defined type for use in the completion handler.

Execution is as follows:
1. Task 0 initializes a data buffer and sends it to Task 1 with a LAPI_Amsend, then waits on its own flag (not a counter managed by LAPI).
2. The arrival of the first packet on Task 1 causes LAPI to invoke Task 1’s header handler (defined in the sample source code). The header handler sets up the data structure to pass to the completion handler, sets the completion handler pointer and completion handler parameter pointer, then returns an address in Task 1’s address space.

LAPI uses this address as the base address for writing the transferred data. Note that a header handler definition is required, since it is the means by which LAPI gets the base address for writing. If a completion handler is to be used, the header handler is also where the completion handler pointer is set. Note that a completion handler is optional. If one is not used, the completion handler pointer should be set to NULL to ensure that LAPI does not interpret an uninitialized pointer as the address of a function.

3. Once all data has been transferred, LAPI invokes the completion handler (also defined in the sample source code). The completion handler in this case performs the data computation, using the values passed through the completion handler parameter, and then completes a LAPI_Amsend call back to the original sender. This call invokes the same sequence of steps as above, but on the opposite task.
4. Task 0’s header handler is invoked upon the arrival of the first data packet. The completion handler pointer and parameter pointer are set and a buffer address in Task 0’s address space is returned.
5. Upon completion of the data transfer, Task 0’s completion handler is run. The final step of the completion handler is to increment the flag on which Task 0’s main execution path has been waiting since immediately after the original LAPI_Amsend call. This frees Task 0 to drop into the final fence and then cleanup and terminate.

Figure 21 on page 294 illustrates the sequence of execution in program sample accumulate_and_return.Am.
accumulate_and_return.Put.c

accumulate_and_return.Put.c uses LAPI_Put to do both transfers. Target counters are used for synchronization on both sides. The execution sequence is represented in Figure 21. Task 0 (or any driver task) begins by issuing a LAPI_Put to its buddy (Task 1 in the diagram) then immediately goes into a wait on Target Counter 0. Task 1, meanwhile has initiated a wait on Target Counter 1. Once the LAPI_Put from Task 0 completes on Task 1, Target Counter 1 will increment, freeing Task 1 from its wait.

Task 1 then does the accumulate and returns the data to Task 0 with another LAPI_Put. The completion of this LAPI_Put causes Target Counter 0 to increment, releasing Task 0 from its wait. Both tasks then sync in a final fence before implementing cleanup operations and terminating.

Figure 21 illustrates the sequence of execution in program sample accumulate_and_return.Put.

accumulate_and_return.Xfer.c

accumulate_and_return.Xfer.c executes the same sequence as
accumulate_and_return.Put except that the LAPI_Xfer interface is used for the first put call. For additional examples of using the LAPI_Xfer interface, see the sample programs in the xfer directory.

**vector**
Contains illustrations of LAPI’s vector interface. accumulate_and_return.Amv.c uses LAPI_Amsendv to perform the same set of tasks as the samples in the basic directory. matrix.c demonstrates a two-dimensional data transfer using LAPI vectors, and strided.c illustrates a strided vector transfer.

**dgsp**
Provides samples illustrating LAPI’s DGSP interface. Dgsp_simple.c builds a DGSP and then runs a data transfer based on it. The data is unpacked sequentially on the receive side. Dgsp_scatter.c performs the same function, except that the DGSP is on the scatter side.

**xfer**
Illustrates the LAPI_Xfer interface. Am_xfer.c illustrates the use of LAPI_Xfer to do an equivalent communication to a LAPI_Amsend call. Put_Xfer.c illustrates the equivalent of a LAPI_Put call. Put_Xfer.F illustrates the equivalent of a LAPI_PUT call. For HPS systems running RSCT LAPI for AIX 5.3 or 6.1, Hw_xfer.c illustrates the use of LAPI_Xfer for RDMA operations.

The remaining directories focus on special cases of LAPI operation:

**interop**
Demonstrates the use of the LAPI_Xfer interface for interoperability between 32-bit and 64-bit LAPI applications. Using LAPI_Address_init64 and LAPI_Xfer calls allows remote addresses to be exchanged as 64-bit values in either case. The build instructions are more complicated in this example. A 32-bit and 64-bit executable must be built and run together to truly demonstrate this interoperability. Scripts are provided for convenience in building. See the interop/README.LAPI.INTEROP file for details.

**standalone**
Illustrates methods for running RSCT LAPI for AIX or LAPI for Linux in standalone mode, that is, without the use of PE and TWS LoadLeveler. Setup for standalone mode is slightly different for user space (US) mode than it is for UDP mode. A separate example “subgroup” is created for each case.

- Standalone UDP initialization involves providing a means for distributing task address and port information that would normally be distributed by PE. LAPI supports two methods for distributing this information, the use of a user handler, and the use of a user list. Building and execution of files is different than in the usual LAPI environment. Binaries must be created using a non-parallel compiler. Also, certain environment variables that are normally set by PE must be set by hand for each task before executing. Finally, each task must be executed by hand. See the standalone/udp/README.LAPI.STANDALONE.UDP file for details on building and execution for standalone UDP operation.

- For HPS systems running RSCT LAPI for AIX 5.3 or 6.1, standalone US initialization involves the setting of certain environment variables as well as the execution of each task by hand. The user must also reserve an adapter window for each task and load the network tables on each node. The user must then use the adapter and window information to set the **MP_LAPI_NETWORK** environment variable before execution of each task. A set of helper applications for loading network tables and grabbing network information are provided in the standalone/us/ntbl directory. Note that these applications require a system administrator for building and installation. Full
details on building and executing for standalone user space applications can be found in the standalone/us/README.LAPI.STANDALONE.US file.

Using the LAPI sample programs

The LAPI sample programs are arranged to provide a threaded tutorial for new users, as well as a reference for both experienced and new LAPI users. Users already familiar with LAPI can dive right into the samples anywhere they wish, exploring directories such as dgsp, vector or xfer to learn about specific LAPI constructs or techniques. The sample programs in the init subdirectory may be a useful reference for experienced users, especially the Qenv_senv sample, which illustrates all runtime parameters that can be queried using LAPI_Qenv and set using LAPI_Senv. The sample also prints the default values of each of the parameters. The lapi_api directory is also a handy reference tool, because it provides a sample for each LAPI communication API call.

IBM suggests that new users:
1. Start with the init subdirectory to get a feel for the basics of writing, building and executing LAPI applications, as well as the parameters and API calls that come into play during setup. The execution model is single task for these examples, making them easier to understand for starters.
2. Go through some of the samples in the lapi_api directory to understand LAPI’s communication APIs.
3. View the remaining directories in any order to demonstrate specific aspects of the LAPI API.

The interop and standalone directories are special cases for users who have 32-bit and 64-bit LAPI applications that need to communicate (see the interop directory) or for users who are running without PE or TWS LoadLeveler (see the standalone directory).

Summary of constructs and techniques for LAPI programming

<table>
<thead>
<tr>
<th>For examples of this LAPI programming construct or technique:</th>
<th>See this sample:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Origin counter</td>
<td>Vector: strided.c</td>
</tr>
<tr>
<td>Target counter</td>
<td>Lapi_api: Put.c, Putf.F, Putv.c, Putvf.F</td>
</tr>
<tr>
<td></td>
<td>Basic: accumulate_and_return.Put.c, accumulate_and_return.Xfer.c</td>
</tr>
<tr>
<td></td>
<td>Xfer: Put_xfer.c, Put_xferf.F</td>
</tr>
<tr>
<td>Completion counter</td>
<td>Lapi_api: Am.c, Amv.c, Putv.c, Putvf.F</td>
</tr>
<tr>
<td></td>
<td>Dgsp: Dgsp_simple.c</td>
</tr>
<tr>
<td></td>
<td>Vector: accumulate_and_return.Amv.c, matrix.c, strided.c</td>
</tr>
<tr>
<td></td>
<td>Xfer: Am_xfer.c, Put_xferf.c</td>
</tr>
<tr>
<td>Default values of LAPI runtime parameters</td>
<td>Init: Qenv_senv.c</td>
</tr>
<tr>
<td>Setting of LAPI runtime parameters</td>
<td>Init: Qenv_senv.c</td>
</tr>
<tr>
<td>User Error handler</td>
<td>Init: Init.c</td>
</tr>
</tbody>
</table>
Table 56. Constructs and techniques for LAPI programming (continued)

<table>
<thead>
<tr>
<th>For examples of this LAPI programming construct or technique:</th>
<th>See this sample:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization of a LAPI handle after termination</td>
<td>Init: Init.c</td>
</tr>
<tr>
<td>Address Manipulation in LAPI</td>
<td>Init: Addr.c</td>
</tr>
<tr>
<td>User header data</td>
<td>Basic: accumulate_and_return.Am.c vector: accumulate_and_return.Amv.c</td>
</tr>
<tr>
<td>Completion handler parameter</td>
<td>Basic: accumulate_and_return.Am.c Vector: accumulate_and_return.Amv.c</td>
</tr>
<tr>
<td>LAPI communication API calls from within a completion handler</td>
<td>Basic: accumulate_and_return.Am.c Vector: accumulate_and_return.Amv.c</td>
</tr>
</tbody>
</table>
Appendix A. Accessibility features for RSCT

Accessibility features help users who have a disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility features

The following list includes the major accessibility features in RSCT:

- Keyboard-only operation
- Interfaces that are commonly used by screen readers
- Keys that are discernible by touch but do not activate just by touching them
- Industry-standard devices for ports and connectors
- The attachment of alternative input and output devices

The IBM Cluster information center and its related publications are accessibility-enabled. The accessibility features of the information center are described in the Accessibility and keyboard shortcuts in the information center topic at:

http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp

Related accessibility information

You can view these publications in Adobe Portable Document Format (PDF) using the Adobe Acrobat Reader. The PDFs are available from the IBM Cluster information center:

http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp

and the IBM Publications Center:

http://www.ibm.com/shop/publications/order

IBM and accessibility

See the IBM Human Ability and Accessibility Center for more information about the commitment that IBM has to accessibility:

http://www.ibm.com/able
Appendix B. Product-related information

You can use the low-level application programming interface (LAPI) “standalone” or with the IBM Parallel Environment (PE) licensed program. Specifically, you can use the parallel operating environment (POE) component of PE to compile and run LAPI parallel programs. Though the use of PE is optional and requires additional steps, it is recommended that you use PE with LAPI. Unless otherwise noted, this book discusses the use of LAPI in conjunction with PE.

For AIX 6.1, the current version of LAPI is a subcomponent of the Reliable Scalable Cluster Technology (RSCT) component of the AIX 6 operating system.

For AIX Version 5.3, the current version of LAPI is a subcomponent of the RSCT component of the AIX 5L operating system.

For Linux, the current version of LAPI is packaged with PE.

For information about:
- installing PE, see PE: Installation
- using POE, see PE: Operation and Use, Volume 1
- AIX, go to: http://www.ibm.com/servers/aix/library
- RSCT, see the RSCT: Administration Guide
- using LAPI without PE, see Chapter 17, “Using LAPI on a standalone system,” on page 141

LAPI version

This edition applies to:
- version 3.1.3.0 of LAPI for Linux
- version 3.1.4.0 of RSCT LAPI for AIX 6.1
- version 2.4.7.0 of RSCT LAPI for AIX 5.3

To find out which version of LAPI is running on a particular AIX node, enter:

dlpp -L rsct.lapi.*

The output will look like this:

<table>
<thead>
<tr>
<th>Fileset</th>
<th>Level</th>
<th>State</th>
<th>Type</th>
<th>Description (Uninstaller)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rsct.lapi.rte</td>
<td>3.1.4.0</td>
<td>C</td>
<td>F</td>
<td>RSCT LAPI</td>
</tr>
</tbody>
</table>

To find out which version of LAPI is running on a particular Linux node, enter:

rpm -qa | grep lapi

The output will look like this:

| lapi_x86_32bit_base_IP_rh500-3.1.3.0-0611a |
| lapi_x86_64bit_IP_rh500-3.1.3.0-0611a    |
| lapi_x86_32bit_US_rh500-3.1.3.0-0611a    |
| lapi_x86_64bit_US_rh500-3.1.3.0-0611a    |
Case sensitivity in AIX

Everything in the AIX operating system is case-sensitive, which means that it distinguishes between uppercase and lowercase letters. For example, you can use the `ls` command to list files. If you type `LS`, the system responds that the command is not found. Likewise, `FILEA`, `FileA`, and `filea` are three distinct file names, even if they reside in the same directory. To avoid causing undesirable actions to be performed, always make sure you use the correct case.

ISO 9000

ISO 9000 registered quality systems were used in the development and manufacturing of this product.

Product-related feedback

To contact the IBM cluster development organization, send your comments by e-mail to:

`cluster@us.ibm.com`
Appendix C. LAPI execution models

LAPI provides two different execution models:
1. the Internet Protocol / user space (IP/US) execution model
2. the shared memory execution model

The IP/US execution model

LAPI provides a thread-safe environment and supports an execution model that allows for execution concurrency between LAPI instances and LAPI client applications.

Using the setup function (LAPI_Init), a user process establishes a LAPI context. Within a LAPI context, the LAPI library is thread-safe and multiple threads can make LAPI calls within the same context. The different calls can run concurrently with each other and with the user threads. In reality, however, execution concurrency among these calls is limited by the locking that is required with LAPI to maintain integrity of its internal data structures and the need to share a single underlying communication channel.

As with any multi-threaded application, coherence of user data is the responsibility of the user. Specifically, if two or more LAPI calls from different threads can run concurrently and if they specify overlapping user buffer areas, the result is undefined. It is the responsibility of the user to coordinate the required synchronization between threads that operate on overlapping buffers.

The user application thread, as well as the completion handlers, cannot hold mutual exclusion resources before making LAPI calls; if they do, it is possible to run into deadlock situations.

Because user-defined handlers can be called concurrently from multiple threads, it is the user’s responsibility to make them thread-safe.

Figure 23 on page 304 shows the interaction among an application thread, an interrupt/timer thread, and a completion thread.
Whenever possible, thread 0 (the application thread) and thread 1 (the interrupt/timer thread) try to call the LAPI dispatcher. This way, progress on incoming and outgoing messages can be made while minimizing additional overhead. Most LAPI calls that are made by the application thread also result in the LAPI dispatcher being run automatically. The interrupt/timer thread waits in the kernel for the occurrence of a notification event. When an event occurs, the kernel "wakes up" the waiting thread. As shown in Figure 23, after the interrupt/timer thread returns from waiting in the kernel, it calls the LAPI dispatcher.

The LAPI dispatcher is the central control point that orchestrates the invocation of the functions and threads needed to process outstanding incoming and outgoing LAPI messages. The LAPI dispatcher can run from the application thread, the interrupt/timer thread, or the completion thread. To maintain integrity, locking is used to ensure that only one instance of the dispatcher runs at a time. On incoming messages, the LAPI dispatcher manages the reassembly of data from different packets — which might arrive out-of-order — into the specified buffer, and then calls the completion handler if necessary.

LAPI_Init creates thread 2 to run completion handlers that are associated with active messages. User-written completion handlers can make LAPI function calls that in turn call the LAPI dispatcher. The completion handler thread processes work from the completion handler queue. When the queue is empty, the thread waits using a pthread_cond_wait(). If an active message (LAPI_Amsend) includes a completion handler, the dispatcher queues a request on the completion queue after the whole message has arrived and has been reassembled in the specified buffer. The dispatcher then sends a pthread_cond_signal to the completion handler thread. If this thread was in a wait state, it will begin processing the completion handler queue; otherwise, if it was not waiting, the thread signal is ignored.

Figure 23. A LAPI thread model
LAPI handlers are not guaranteed to run one at a time. LAPI calls can run concurrently within the origin, the target, or both. The restriction about not holding on to mutual exclusion resources when making LAPI calls still applies.

This discussion of a thread-safe environment and execution concurrence within the LAPI library applies to both polling mode and interrupt mode. In polling mode, any calls to the communication library try to make progress on the context specified in the call. LAPI includes the \texttt{LAPI\_Probe} subroutine, which lets applications explicitly check for and handle incoming messages.

The execution model of the handlers consists of the following events:

<table>
<thead>
<tr>
<th>Event</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message arrival</td>
<td>Copies the message from the network into the appropriate data access memory space.</td>
</tr>
<tr>
<td>Interrupt or poll</td>
<td>Causes an interrupt if required, based on the mode.</td>
</tr>
<tr>
<td>Dispatcher start</td>
<td>Calls LAPI's internal communication dispatcher.</td>
</tr>
<tr>
<td>New message packet</td>
<td>Checks the LAPI header and determines (by checking the receive state message reassembly table) whether the packet is part of a pending message or whether it is a new message. For the first packet of a new message, calls the header handler function.</td>
</tr>
<tr>
<td>Return from header handler</td>
<td>If the message is contained in more than one packet, the LAPI dispatcher logs that there is a pending message, saves the completion handler address, and saves the user's buffer address to be used during the message reassembly of pending message packets.</td>
</tr>
<tr>
<td>Pending message packet</td>
<td>Copies the message to the appropriate portion of the user buffer specified through the header handler. If the packet completes the message, the dispatcher queues the completion handler; otherwise, the dispatcher returns to check for message arrivals.</td>
</tr>
<tr>
<td>Return from completion handler</td>
<td>When the completion handler is run, it updates the appropriate target counter before continuing.</td>
</tr>
</tbody>
</table>

**The shared memory execution model**

When tasks are on the same node, it is more efficient for communication protocol clients to use a shared memory protocol, as opposed to using the switch adapter or "double copy" mode (communication through a shared segment). Using a shared memory protocol reduces switch congestion and optimizes performance. This is where LAPI's shared memory execution model comes in handy.

\textbf{Figure 24 on page 306} and \textbf{Figure 25 on page 306} show the different approaches for running \texttt{LAPI\_Put} when task 0 and task 1 are on the same node. The user interface is identical in the LAPI communication paths for shared memory and for nonshared memory.
LAPI shared memory

Each task that communicates with other tasks using shared memory has a message queue of command structures, called slots. Shared memory task \( n \) processes commands off of the head of the task \( n \) message queue. Tasks other than task \( n \) update the tail of the task \( n \) message queue. Each slot contains a fixed-size data area.

There are two ways to transfer data using shared memory:

1. For small messages, the message is copied by the message origin into one or more slots at the tail of the target task’s message queue. The message is subsequently copied out of these slots by the target task and copied into the user buffers at the target task. This message transfer mode is called slot mode and this flow is referred to as the slot flow.

2. For large messages, the virtual memory region that contains the message data at the source task is exported using a kernel extension. The kernel extension returns identifiers to the source task. These identifiers can be used to attach to
the source task’s virtual memory region. The source task transfers these identifiers to the target task using a slot at the tail of the target task’s message queue. When this slot is processed on the message target, the identifiers are used to attach the source message region to the target task’s address space, and to thereafter copy the data directly from the source task message buffers into user buffers at the target task. This message transfer mode is called attach mode and this flow is referred to as the attach flow.

Requirements and considerations

- To use shared memory, set the (non-case-sensitive) LAPI_USE_SHM environment variable to yes or only.
  
  If LAPI_USE_SHM=only, LAPI only uses the shared memory mechanism. If all tasks are not on the same node or if shared memory setup is not successful, LAPI returns an error message.

  If LAPI_USE_SHM=yes, the shared memory path is used on tasks within a node, unless initialization of shared memory fails. In that case, the switch path is used. For tasks on different nodes, the network path is used.

- The kernel extension must be loaded.

- The maximum number of shared memory tasks per operating system image is 128.

- If you are using LAPI shared memory on Linux (running over IP or US), you should increase the maximum shared memory size to 256MB. To do this, enter:
  
  `echo 268435456 > /proc/sys/kernel/shmmax`

Cross-memory kernel extension on AIX

During AIX initialization, a kernel extension that supports LAPI’s shared memory execution model is loaded. This kernel extension allows one task to export a portion of its address space to another task of the parallel application. The exported portion of the address space is attached to the address space of the companion task and the data transfer is done by a simple copy. A separate shared-memory region is required for exchanging metadata that describes the exported regions and allows for handshaking between the communicating tasks of the parallel application. This shared-memory region can also be used for transferring small messages or certain types of noncontiguous messages.

As part of rsct.lapi installation, the configuration method for the kernel extension is added to the Config_Rules object data manager (ODM) database, to be run in phase 3 of the system initialization. This ensures that the kernel extension is available for any rsct.lapi installation, whether a switch adapter is present or not. See Chapter 4, “Installing RSCT LAPI for AIX,” on page 29 for more information.
Appendix D. LAPI messages, return codes, and return values

This appendix lists LAPI’s attention messages, error codes, return codes, and return values. For LAPI error messages, see RSCT: Messages.

LAPI attention messages

Table 57. LAPI attention messages

<table>
<thead>
<tr>
<th>Attention message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>485</td>
<td>Failover setup failed because an internal error occurred.</td>
</tr>
<tr>
<td>486</td>
<td>Failover setup failed because NAM is not installed.</td>
</tr>
<tr>
<td>487</td>
<td>Failover setup failed because group services is not installed.</td>
</tr>
<tr>
<td>488</td>
<td>Failover setup failed because an old version of POE is installed.</td>
</tr>
<tr>
<td>489</td>
<td>Failover setup failed because POE is not installed.</td>
</tr>
<tr>
<td>490</td>
<td>Failover setup failed because a non-snX device is being used by the current job.</td>
</tr>
<tr>
<td>491</td>
<td>Failover function halted due to an internal LAPI error.</td>
</tr>
<tr>
<td>499</td>
<td>Bulk transfer is enabled.</td>
</tr>
<tr>
<td>500</td>
<td>Timeout between multiple tasks.</td>
</tr>
<tr>
<td>501</td>
<td>LAPI version string.</td>
</tr>
<tr>
<td>502</td>
<td>Shared memory initialization failed.</td>
</tr>
<tr>
<td>503</td>
<td>Shared memory initialization failed at checkpoint restart.</td>
</tr>
<tr>
<td>504</td>
<td>Shared memory was not used because only one task is running.</td>
</tr>
<tr>
<td>505</td>
<td>The task was not able to create shared memory.</td>
</tr>
<tr>
<td>506</td>
<td>The task was not able to get shared memory.</td>
</tr>
<tr>
<td>507</td>
<td>The task was not able to attach shared memory.</td>
</tr>
<tr>
<td>508</td>
<td>The task was not able to reserve a segment.</td>
</tr>
<tr>
<td>509</td>
<td>The initial instance used for striping communication.</td>
</tr>
<tr>
<td>520</td>
<td>Preemption setup failed. (AIX)</td>
</tr>
</tbody>
</table>

LAPI return codes

Table 58. LAPI return codes

<table>
<thead>
<tr>
<th>Return code</th>
<th>Return values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LAPI_SUCCESS</td>
<td>The call completed successfully.</td>
</tr>
</tbody>
</table>

LAPI error codes

Table 59 on page 310 lists all of the LAPI error codes and their associated return values in numerical order (by error code).
<table>
<thead>
<tr>
<th>Error code</th>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>LAPI_ERR_UNKNOWN</td>
<td>An asynchronous, internal communication error occurred.</td>
</tr>
<tr>
<td>401</td>
<td>LAPI_ERR_ALL_HNDL_IN_USE</td>
<td>All available LAPI instances are in use.</td>
</tr>
<tr>
<td>402</td>
<td>LAPI_ERR_BOTH_NETSTR_SET</td>
<td>Both network statements are set for a single LAPI instance.</td>
</tr>
<tr>
<td>404</td>
<td>LAPI_ERR_CSS_LOAD_FAILED</td>
<td>Unable to load the communication utility library.</td>
</tr>
<tr>
<td>405</td>
<td>LAPI_ERR_INFO_NULL</td>
<td>The lapi_info pointer is NULL.</td>
</tr>
<tr>
<td>406</td>
<td>LAPI_ERR_MSG_API</td>
<td>The MP_MSG_API environment setting has an error in it.</td>
</tr>
<tr>
<td>408</td>
<td>LAPI_ERR_NO_UDP_HNDLR</td>
<td>Told LAPI to use a user-defined udp_hndlr, but udp_hndlr is set to NULL.</td>
</tr>
<tr>
<td>409</td>
<td>LAPI_ERR_HDR_HNDLR_NULL</td>
<td>The header handler is NULL. This error is returned in an asynchronous error handler.</td>
</tr>
<tr>
<td>410</td>
<td>LAPI_ERR_PSS_NON_ROOT</td>
<td>Tried to initialize the persistent subsystem (PSS) as non-root.</td>
</tr>
<tr>
<td>412</td>
<td>LAPI_ERR_SHM_CE_NOT_LOADED</td>
<td>The shared memory kernel extension is not loaded.</td>
</tr>
<tr>
<td>413</td>
<td>LAPI_ERR_TIMEOUT</td>
<td>A communication timeout has occurred. This error is returned in an asynchronous error handler.</td>
</tr>
<tr>
<td>414</td>
<td>LAPI_ERR_REG_TIMER</td>
<td>An error occurred while re-registering the timer.</td>
</tr>
<tr>
<td>415</td>
<td>LAPI_ERR_UDP_PKT_SZ</td>
<td>The UDP packet size is not valid.</td>
</tr>
<tr>
<td>416</td>
<td>LAPI_ERR_USER_UDP_HNDLR_FAIL</td>
<td>The user-defined udp_hndlr failed.</td>
</tr>
<tr>
<td>417</td>
<td>LAPI_ERR_HNDL_INVALID</td>
<td>A non-valid handle was passed in to LAPI.</td>
</tr>
<tr>
<td>418</td>
<td>LAPI_ERR_RET_PTR_NULL</td>
<td>The output data pointer is NULL.</td>
</tr>
<tr>
<td>419</td>
<td>LAPI_ERR_ADDR_HNDL_RANGE</td>
<td>The address handle range is not valid.</td>
</tr>
<tr>
<td>420</td>
<td>LAPI_ERR_ADDR_TBL_NULL</td>
<td>The output address table is NULL.</td>
</tr>
<tr>
<td>421</td>
<td>LAPI_ERR_TGT_PURGED</td>
<td>The destination task is purged.</td>
</tr>
<tr>
<td>422</td>
<td>LAPI_ERR_MULTIPLE_WAITERS</td>
<td>Multiple threads are waiting for the same counter.</td>
</tr>
<tr>
<td>423</td>
<td>LAPI_ERR_MEMORY_EXHAUSTED</td>
<td>LAPI is unable to allocate storage.</td>
</tr>
<tr>
<td>424</td>
<td>LAPI_ERR_INFO_NONZERO</td>
<td>Unused fields in the lapi_info_t structure need to be zeroed out.</td>
</tr>
<tr>
<td>425</td>
<td>LAPI_ERR_ORG_ADDR_NULL</td>
<td>The source address pointer is NULL.</td>
</tr>
<tr>
<td>426</td>
<td>LAPI_ERR_TGT_ADDR_NULL</td>
<td>The target address pointer is NULL.</td>
</tr>
<tr>
<td>427</td>
<td>LAPI_ERR_DATA_LEN</td>
<td>The length passed in is too big.</td>
</tr>
<tr>
<td>428</td>
<td>LAPI_ERR_TGT</td>
<td>The target is not valid.</td>
</tr>
<tr>
<td>429</td>
<td>LAPI_ERR_UHDR_NULL</td>
<td>uhdr is NULL, but uhdr_len is greater than 0.</td>
</tr>
<tr>
<td>430</td>
<td>LAPI_ERR_UHDR_LEN</td>
<td>uhdr_len is too big.</td>
</tr>
<tr>
<td>431</td>
<td>LAPI_ERR_HDR_LEN</td>
<td>uhdr_len is not word-aligned.</td>
</tr>
<tr>
<td>432</td>
<td>LAPI_ERR_ORG_EXTENT</td>
<td>The source vector's extent is too big.</td>
</tr>
<tr>
<td>433</td>
<td>LAPI_ERR_ORG_STRIDE</td>
<td>The source vector's stride is less than its block.</td>
</tr>
<tr>
<td>434</td>
<td>LAPI_ERR_NO_CONNECTIVITY</td>
<td>No connectivity to task.</td>
</tr>
<tr>
<td>Error code</td>
<td>Return value</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>435</td>
<td>LAPI_ERR_ADAPTERS_DOWN</td>
<td>All adapters are down.</td>
</tr>
<tr>
<td>436</td>
<td>LAPI_ERR_RECV_INCOMP</td>
<td>The &quot;message receive&quot; operation did not complete.</td>
</tr>
<tr>
<td>437</td>
<td>LAPI_ERR_SEND_INCOMP</td>
<td>The &quot;message send&quot; operation did not complete.</td>
</tr>
<tr>
<td>438</td>
<td>LAPI_ERR_SEND_TIMEOUT</td>
<td>The &quot;message send&quot; operation timed out.</td>
</tr>
<tr>
<td>439</td>
<td>LAPI_ERR_SHM_SETUP</td>
<td>The shared memory setup failed.</td>
</tr>
<tr>
<td>440</td>
<td>LAPI_ERR_ORG_VEC_ADDR</td>
<td>The source vector address is NULL, but its \texttt{len} is greater than \texttt{0}.</td>
</tr>
<tr>
<td>441</td>
<td>LAPI_ERR_ORG_VEC_LEN</td>
<td>The source vector's length is too big.</td>
</tr>
<tr>
<td>442</td>
<td>LAPI_ERR_ORG_VEC_NULL</td>
<td>The source vector pointer is NULL.</td>
</tr>
<tr>
<td>443</td>
<td>LAPI_ERR_ORG_VEC_TYPE</td>
<td>The source vector type is not valid.</td>
</tr>
<tr>
<td>444</td>
<td>LAPI_ERR_STRIDE_ORG_VEC_ADDR_NULL</td>
<td>The source stride vector address is NULL.</td>
</tr>
<tr>
<td>445</td>
<td>LAPI_ERR_STRIDE_TGT_VEC_ADDR_NULL</td>
<td>The target stride vector address is NULL.</td>
</tr>
<tr>
<td>446</td>
<td>LAPI_ERR_TGT_EXTENT</td>
<td>The target vector's extent is too big.</td>
</tr>
<tr>
<td>447</td>
<td>LAPI_ERR_TGT_STRIDE</td>
<td>The target vector's stride is less than its block.</td>
</tr>
<tr>
<td>448</td>
<td>LAPI_ERR_TGT_VEC_ADDR</td>
<td>The target vector address is NULL, but its \texttt{len} is greater than \texttt{0}.</td>
</tr>
<tr>
<td>449</td>
<td>LAPI_ERR_TGT_VEC_LEN</td>
<td>The target vector's length is too big.</td>
</tr>
<tr>
<td>451</td>
<td>LAPI_ERR_TGT_VEC_NULL</td>
<td>The target vector pointer is NULL.</td>
</tr>
<tr>
<td>452</td>
<td>LAPI_ERR_TGT_VEC_TYPE</td>
<td>The target vector type is not valid.</td>
</tr>
<tr>
<td>453</td>
<td>LAPI_ERR_VEC_NUM_DIFF</td>
<td>The source and target vectors have different \texttt{num_vecs} values.</td>
</tr>
<tr>
<td>454</td>
<td>LAPI_ERR_VEC_TYPE_DIFF</td>
<td>The source and target vectors have different \texttt{vec_type} values.</td>
</tr>
<tr>
<td>455</td>
<td>LAPI_ERR_VEC_LEN_DIFF</td>
<td>The source and target vectors have different \texttt{len[]} values.</td>
</tr>
<tr>
<td>456</td>
<td>LAPI_ERR_MSG_INFO_NULL</td>
<td>LAPI_Msgpoll's \texttt{info} pointer is NULL.</td>
</tr>
<tr>
<td>458</td>
<td>LAPI_ERR_CNTR_NULL</td>
<td>The counter pointer is NULL.</td>
</tr>
<tr>
<td>459</td>
<td>LAPI_ERR_CNTR_VAL</td>
<td>The counter value passed in is less than \texttt{0} for the LAPI_Nopoll_wait call.</td>
</tr>
<tr>
<td>460</td>
<td>LAPI_ERR_QUERY_TYPE</td>
<td>The query is not a valid query type.</td>
</tr>
<tr>
<td>461</td>
<td>LAPI_ERR_IN_VAL_NULL</td>
<td>LAPI_Rmw's \texttt{in_val} pointer is NULL.</td>
</tr>
<tr>
<td>462</td>
<td>LAPI_ERR_RMW_OP</td>
<td>The RMW operator is not valid.</td>
</tr>
<tr>
<td>463</td>
<td>LAPI_ERR_TGT_VAR_NULL</td>
<td>LAPI_Rmw's \texttt{tgt_var} address is NULL.</td>
</tr>
<tr>
<td>464</td>
<td>LAPI_ERR_SET_VAL</td>
<td>LAPI_Senv's \texttt{set_val} value is not valid.</td>
</tr>
<tr>
<td>465</td>
<td>LAPI_ERR_DGSP</td>
<td>The DGSP is NULL or is not registered.</td>
</tr>
<tr>
<td>466</td>
<td>LAPI_ERR_DGSP_ATOM</td>
<td>The DGSP \texttt{atom_size} is not valid.</td>
</tr>
<tr>
<td>467</td>
<td>LAPI_ERR_DGSP_BRANCH</td>
<td>The DGSP processed an incorrect branch.</td>
</tr>
<tr>
<td>468</td>
<td>LAPI_ERR_DGSP_CTL</td>
<td>The DGSP CONTROL instruction has errors.</td>
</tr>
<tr>
<td>469</td>
<td>LAPI_ERR_DGSP_COPY_SZ</td>
<td>The DGSP has a non-valid copy length.</td>
</tr>
<tr>
<td>470</td>
<td>LAPI_ERR_DGSP_FREE</td>
<td>A non-valid attempt was made to free a DGSP.</td>
</tr>
<tr>
<td>471</td>
<td>LAPI_ERR_DGSP_OPCODE</td>
<td>The DGSP \texttt{opcode} is not valid.</td>
</tr>
<tr>
<td>Error code</td>
<td>Return value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>472</td>
<td>LAPI_ERR_DGSP_REPS</td>
<td>The DGSP has a non-valid reps field (its value is less than 0).</td>
</tr>
<tr>
<td>473</td>
<td>LAPI_ERR_DGSP_STACK</td>
<td>An insufficient stack depth was allocated for the DGSP stack.</td>
</tr>
<tr>
<td>474</td>
<td>LAPI_ERR_OP_SZ</td>
<td>The lapi_rmw_t size is not set to 32 or 64.</td>
</tr>
<tr>
<td>475</td>
<td>LAPI_ERR_UDP_PORT_INFO</td>
<td>The udp_port information pointer is NULL.</td>
</tr>
<tr>
<td>476</td>
<td>LAPI_ERR_XFER_CMD</td>
<td>The LAPI_Xfer command type is not valid.</td>
</tr>
<tr>
<td>477</td>
<td>LAPI_ERR_UTIL_CMD</td>
<td>The LAPI_Util command type is not valid.</td>
</tr>
<tr>
<td>478</td>
<td>LAPI_ERR_CATALOG_FAIL</td>
<td>LAPI cannot open the message catalog.</td>
</tr>
<tr>
<td>479</td>
<td>LAPI_ERR_PACK_SZ</td>
<td>The pack buffer is too small.</td>
</tr>
<tr>
<td>480</td>
<td>LAPI_ERR_DGSP_OTHER</td>
<td>A DGSP error occurred (code_size is equal to 0, for example).</td>
</tr>
<tr>
<td>481</td>
<td>LAPI_ERR_UDP_SOCKET</td>
<td>An error occurred during a UDP socket operation.</td>
</tr>
<tr>
<td>482</td>
<td>LAPI_ERR_COLLECTIVE_PSS</td>
<td>The persistent subsystem (PSS) attempted a collective call.</td>
</tr>
<tr>
<td>492</td>
<td>LAPI_ERR_TGT_CONTEXT</td>
<td>Non-valid target context.</td>
</tr>
<tr>
<td>493</td>
<td>LAPI_ERR_SRC_CONTEXT</td>
<td>Non-valid source context.</td>
</tr>
<tr>
<td>494</td>
<td>LAPI_ERR_TGT_BUFHNDL</td>
<td>Non-valid target buffer handle.</td>
</tr>
<tr>
<td>495</td>
<td>LAPI_ERR_SRC_BUFHNDL</td>
<td>Non-valid source buffer handle.</td>
</tr>
<tr>
<td>496</td>
<td>LAPI_ERR_TGT_OFFSET</td>
<td>Non-valid target offset.</td>
</tr>
<tr>
<td>497</td>
<td>LAPI_ERR_SRC_OFFSET</td>
<td>Non-valid source offset.</td>
</tr>
<tr>
<td>498</td>
<td>LAPI_ERR_NO_RDMARESOURCE</td>
<td>No RDMA resources.</td>
</tr>
<tr>
<td>499</td>
<td>LAPI_ERR_NO_RDMARESOURCE</td>
<td>No RDMA resources.</td>
</tr>
<tr>
<td>510</td>
<td>LAPI_ERR_NO_ENV_VAR</td>
<td>A required environment variable is not set.</td>
</tr>
<tr>
<td>511</td>
<td>LAPI_ERR_NO_CALLBACK_REG</td>
<td>No callback has been registered.</td>
</tr>
<tr>
<td>512</td>
<td>LAPI_ERR_NO_CNTR_REG</td>
<td>No counter has been registered.</td>
</tr>
<tr>
<td>513</td>
<td>LAPI_ERR_XLATE_FAILED</td>
<td>RDMA translation failed.</td>
</tr>
<tr>
<td>514</td>
<td>LAPI_ERR_RDMA_RESOURCES</td>
<td>There are no more RDMA resources.</td>
</tr>
<tr>
<td>515</td>
<td>LAPI_ERR_LW_DATA_LEN</td>
<td>An incorrect data length, header length, or both were specified for LAPI_AM_LW_XFER.</td>
</tr>
<tr>
<td>516</td>
<td>LAPI_ERR_LW_NO_HNDLR_SET</td>
<td>No header handler has been set for using LAPI_AM_LW_XFER.</td>
</tr>
<tr>
<td>517</td>
<td>LAPI_ERR_OFFSET_LEN</td>
<td>The DGSP send offset is out of range (applies to LAPI_AMX_XFER only).</td>
</tr>
<tr>
<td>518</td>
<td>LAPI_ERR_RCXT_CANCEL</td>
<td>Unable to cancel rCxt.</td>
</tr>
<tr>
<td>519</td>
<td>LAPI_ERR_CODE_UNKNOWN</td>
<td>The error code is unknown to LAPI.</td>
</tr>
<tr>
<td>521</td>
<td>LAPI_ERR_DUP_TRIGGER</td>
<td>The trigger is already registered.</td>
</tr>
<tr>
<td>522</td>
<td>LAPI_ERR_TRIGGER_UNKNOWN</td>
<td>The trigger is not registered.</td>
</tr>
<tr>
<td>523</td>
<td>LAPI_ERR_TASK_NUM</td>
<td>The task number is not correct.</td>
</tr>
<tr>
<td>524</td>
<td>LAPI_ERR_NULL_LIST</td>
<td>The task list is null.</td>
</tr>
<tr>
<td>525</td>
<td>LAPI_ERR_GRP_MEMBER</td>
<td>The group member ID is not correct.</td>
</tr>
</tbody>
</table>
Table 59. LAPI error codes (continued)

<table>
<thead>
<tr>
<th>Error code</th>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>526</td>
<td>LAPI_ERR_NULL_GRP</td>
<td>The group handle is NULL.</td>
</tr>
<tr>
<td>527</td>
<td>LAPI_ERR_GRP</td>
<td>The group handle that was passed is not valid.</td>
</tr>
<tr>
<td>528</td>
<td>LAPI_ERR_STAT_SAVE_FAIL</td>
<td>Failed to upload user’s statistics to the PNSD.</td>
</tr>
</tbody>
</table>

LAPI return values

Table 60 lists all of the LAPI return values and their associated return codes in alphabetical order (by return value).

Table 60. LAPI return values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPI_ERR_ADAPTERS_DOWN</td>
<td>435</td>
<td>All adapters are down.</td>
</tr>
<tr>
<td>LAPI_ERR_ADDR_HNDL_RANGE</td>
<td>419</td>
<td>The address handle range is not valid.</td>
</tr>
<tr>
<td>LAPI_ERR_ADDR_TBL_NULL</td>
<td>420</td>
<td>The output address table is NULL.</td>
</tr>
<tr>
<td>LAPI_ERR_ALL_HNDL_IN_USE</td>
<td>401</td>
<td>All available LAPI instances are in use.</td>
</tr>
<tr>
<td>LAPI_ERR_BOTH_NETSTR_SET</td>
<td>402</td>
<td>Both network statements are set for a single LAPI instance.</td>
</tr>
<tr>
<td>LAPI_ERR_CATALOG_FAIL</td>
<td>478</td>
<td>LAPI cannot open the message catalog.</td>
</tr>
<tr>
<td>LAPI_ERR_CNTR_NULL</td>
<td>458</td>
<td>The counter pointer is NULL.</td>
</tr>
<tr>
<td>LAPI_ERR_CNTR_VAL</td>
<td>459</td>
<td>The counter value passed in is less than 0 for LAPI_Nopoll_wait call.</td>
</tr>
<tr>
<td>LAPI_ERR_CODE_UNKNOWN</td>
<td>519</td>
<td>The error code is unknown to LAPI.</td>
</tr>
<tr>
<td>LAPI_ERR_COLLECTIVE_PSS</td>
<td>482</td>
<td>The persistent subsystem (PSS) attempted a collective call.</td>
</tr>
<tr>
<td>LAPI_ERR_CSS_LOAD_FAILED</td>
<td>404</td>
<td>Unable to load the communication utility library.</td>
</tr>
<tr>
<td>LAPI_ERR_DATA_LEN</td>
<td>427</td>
<td>The length passed in is too big.</td>
</tr>
<tr>
<td>LAPI_ERR_DGSP</td>
<td>465</td>
<td>The DGSP is NULL or is not registered.</td>
</tr>
<tr>
<td>LAPI_ERR_DGSP_ATOM</td>
<td>466</td>
<td>The DGSP atom_size is not valid.</td>
</tr>
<tr>
<td>LAPI_ERR_DGSP_BRANCH</td>
<td>467</td>
<td>The DGSP processed an incorrect branch.</td>
</tr>
<tr>
<td>LAPI_ERR_DGSP_COPY_SZ</td>
<td>469</td>
<td>The DGSP has a non-valid copy length.</td>
</tr>
<tr>
<td>LAPI_ERR_DGSP_CTL</td>
<td>468</td>
<td>The DGSP CONTROL instruction has errors.</td>
</tr>
<tr>
<td>LAPI_ERR_DGSP_FREE</td>
<td>470</td>
<td>A non-valid attempt was made to free a DGSP.</td>
</tr>
<tr>
<td>LAPI_ERR_DGSP_OPCODE</td>
<td>471</td>
<td>The DGSP opcode is not valid.</td>
</tr>
<tr>
<td>LAPI_ERR_DGSP_OTHER</td>
<td>480</td>
<td>A DGSP error occurred (code_size is equal to 0, for example).</td>
</tr>
<tr>
<td>LAPI_ERR_DGSP_REPS</td>
<td>472</td>
<td>The DGSP has a non-valid reps field (its value is less than 0).</td>
</tr>
<tr>
<td>LAPI_ERR_DGSP_STACK</td>
<td>473</td>
<td>An insufficient stack depth was allocated for the DGSP stack.</td>
</tr>
<tr>
<td>LAPI_ERR_DUP_TRIGGER</td>
<td>521</td>
<td>The trigger is already registered.</td>
</tr>
<tr>
<td>LAPI_ERR_GRP</td>
<td>527</td>
<td>The group handle that was passed is not valid.</td>
</tr>
<tr>
<td>LAPI_ERR_GRP_MEMBER</td>
<td>525</td>
<td>The group member ID is not correct.</td>
</tr>
</tbody>
</table>
Table 60. LAPI return values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPI_ERR_HDR_HNDLR_NULL</td>
<td>409</td>
<td>The header handler is NULL. This error is returned in an asynchronous error handler.</td>
</tr>
<tr>
<td>LAPI_ERR_HDR_LEN</td>
<td>431</td>
<td>uhdr_len is not word-aligned.</td>
</tr>
<tr>
<td>LAPI_ERR_HNDL_INVALID</td>
<td>417</td>
<td>A non-valid handle was passed in to LAPI.</td>
</tr>
<tr>
<td>LAPI_ERR_INFO_NONZERO</td>
<td>424</td>
<td>Unused fields in the lapi_info_t structure need to be zeroed out.</td>
</tr>
<tr>
<td>LAPI_ERR_INFO_NULL</td>
<td>405</td>
<td>The lapi_info pointer is NULL.</td>
</tr>
<tr>
<td>LAPI_ERR_IN_VAL_NULL</td>
<td>461</td>
<td>LAPI_Rmw's in_val pointer is NULL.</td>
</tr>
<tr>
<td>LAPI_ERR_LW_DATA_LEN</td>
<td>515</td>
<td>An incorrect data length, header length, or both were specified for LAPI_AM_LW_XFER.</td>
</tr>
<tr>
<td>LAPI_ERR_LW_NO_HNDLR_SET</td>
<td>516</td>
<td>No header handler has been set for using LAPI_AM_LW_XFER.</td>
</tr>
<tr>
<td>LAPI_ERR_MEMORY_EXHAUSTED</td>
<td>423</td>
<td>LAPI is unable to allocate storage.</td>
</tr>
<tr>
<td>LAPI_ERR_MSG_API</td>
<td>406</td>
<td>The MP_MSG_API environment setting has an error in it.</td>
</tr>
<tr>
<td>LAPI_ERR_MSG_INFO_NULL</td>
<td>456</td>
<td>LAPI_Msgpoll's info pointer is NULL.</td>
</tr>
<tr>
<td>LAPI_ERR_MULTIPLE_WAITERS</td>
<td>422</td>
<td>Multiple threads are waiting for the same counter.</td>
</tr>
<tr>
<td>LAPI_ERR_NO_CALLBACK_REG</td>
<td>511</td>
<td>No callback has been registered.</td>
</tr>
<tr>
<td>LAPI_ERR_NO_CNTR_REG</td>
<td>512</td>
<td>No counter has been registered.</td>
</tr>
<tr>
<td>LAPI_ERR_NO_CONNECTIVITY</td>
<td>434</td>
<td>No connectivity to task.</td>
</tr>
<tr>
<td>LAPI_ERR_NO_ENV_VAR</td>
<td>510</td>
<td>A required environment variable is not set.</td>
</tr>
<tr>
<td>LAPI_ERR_NO_UDP_HNDLR</td>
<td>408</td>
<td>Told LAPI to use a user-defined udp_hndlr, but udp_hndlr is set to NULL.</td>
</tr>
<tr>
<td>LAPI_ERR_NULL_GRP</td>
<td>526</td>
<td>The group handle is NULL.</td>
</tr>
<tr>
<td>LAPI_ERR_NULL_LIST</td>
<td>524</td>
<td>The task list is null.</td>
</tr>
<tr>
<td>LAPI_ERR_OFFSET_LEN</td>
<td>517</td>
<td>The DGSP send offset is out of range (applies to LAPI_AMX_XFER only).</td>
</tr>
<tr>
<td>LAPI_ERR_OP_SZ</td>
<td>474</td>
<td>The lapi_rmw_t size is not set to 32 or 64.</td>
</tr>
<tr>
<td>LAPI_ERR_ORG_ADDR_NULL</td>
<td>425</td>
<td>The source address pointer is NULL.</td>
</tr>
<tr>
<td>LAPI_ERR_ORG_EXTENT</td>
<td>432</td>
<td>The source vector's extent is too big.</td>
</tr>
<tr>
<td>LAPI_ERR_ORG_STRIDE</td>
<td>433</td>
<td>The source vector's stride is less than its block.</td>
</tr>
<tr>
<td>LAPI_ERR_ORG_VEC_ADDR</td>
<td>440</td>
<td>The source vector address is NULL, but its len is greater than 0.</td>
</tr>
<tr>
<td>LAPI_ERR_ORG_VEC_LEN</td>
<td>441</td>
<td>The source vector's length is too big.</td>
</tr>
<tr>
<td>LAPI_ERR_ORG_VEC_NULL</td>
<td>442</td>
<td>The source vector pointer is NULL.</td>
</tr>
<tr>
<td>LAPI_ERR_ORG_VEC_TYPE</td>
<td>443</td>
<td>The source vector type is not valid.</td>
</tr>
<tr>
<td>LAPI_ERR_PACK_SZ</td>
<td>479</td>
<td>The pack buffer is too small.</td>
</tr>
<tr>
<td>LAPI_ERR_PSS_NON_ROOT</td>
<td>410</td>
<td>Tried to initialize the persistent subsystem (PSS) as non-root.</td>
</tr>
<tr>
<td>LAPI_ERR_QUERY_TYPE</td>
<td>460</td>
<td>The query is not a valid query type.</td>
</tr>
<tr>
<td>LAPI_ERR_RCXT_CANCEL</td>
<td>518</td>
<td>Unable to cancel rCxt.</td>
</tr>
<tr>
<td>LAPI_ERR_RDMA_RESOURCES</td>
<td>514</td>
<td>There are no more RDMA resources.</td>
</tr>
</tbody>
</table>
Table 60. LAPI return values (continued)

<table>
<thead>
<tr>
<th>Return value</th>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPI_ERR_RECV_INCOMP</td>
<td>436</td>
<td>The &quot;message receive&quot; operation did not complete.</td>
</tr>
<tr>
<td>LAPI_ERR_REG_TIMER</td>
<td>414</td>
<td>An error occurred while re-registering the timer.</td>
</tr>
<tr>
<td>LAPI_ERR_RET_PTR_NULL</td>
<td>418</td>
<td>The output data pointer is NULL.</td>
</tr>
<tr>
<td>LAPI_ERR_RMW_OP</td>
<td>462</td>
<td>The RMW operator is not valid.</td>
</tr>
<tr>
<td>LAPI_ERR_SEND_INCOMP</td>
<td>437</td>
<td>The &quot;message send&quot; operation did not complete.</td>
</tr>
<tr>
<td>LAPI_ERR_SEND_TIMEOUT</td>
<td>438</td>
<td>The &quot;message send&quot; operation timed out.</td>
</tr>
<tr>
<td>LAPI_ERR_SET_VAL</td>
<td>464</td>
<td>LAPI_Senv's set_val value is not valid.</td>
</tr>
<tr>
<td>LAPI_ERR_SHM_KE_NOT_LOADED</td>
<td>412</td>
<td>The shared memory kernel extension is not loaded.</td>
</tr>
<tr>
<td>LAPI_ERR_SHM_SETUP</td>
<td>439</td>
<td>The shared memory setup failed.</td>
</tr>
<tr>
<td>LAPI_ERR_SRC_BUFHNDL</td>
<td>495</td>
<td>Non-valid source buffer handle.</td>
</tr>
<tr>
<td>LAPI_ERR_SRC_CONTEXT</td>
<td>493</td>
<td>Non-valid source context.</td>
</tr>
<tr>
<td>LAPI_ERR_SRC_OFFSET</td>
<td>497</td>
<td>Non-valid source offset.</td>
</tr>
<tr>
<td>LAPI_ERR_STAT_SAVE_FAIL</td>
<td>528</td>
<td>Failed to upload user's statistics to the PNSD.</td>
</tr>
<tr>
<td>LAPI_ERR_STRIDE_ORG_VEC_ADDR_NULL</td>
<td>444</td>
<td>The source stride vector address is NULL.</td>
</tr>
<tr>
<td>LAPI_ERR_STRIDE_TGT_VEC_ADDR_NULL</td>
<td>445</td>
<td>The target stride vector address is NULL.</td>
</tr>
<tr>
<td>LAPI_ERR_TASK_NUM</td>
<td>523</td>
<td>The task number is not correct.</td>
</tr>
<tr>
<td>LAPI_ERR_TGT</td>
<td>428</td>
<td>The target is not valid.</td>
</tr>
<tr>
<td>LAPI_ERR_TGT_ADDR_NULL</td>
<td>426</td>
<td>The target address pointer is NULL.</td>
</tr>
<tr>
<td>LAPI_ERR_TGT_ADDR_BUFHNDL</td>
<td>494</td>
<td>Non-valid target buffer handle.</td>
</tr>
<tr>
<td>LAPI_ERR_TGT_CONTEXT</td>
<td>492</td>
<td>Non-valid target context.</td>
</tr>
<tr>
<td>LAPI_ERR_TGT_EXTENT</td>
<td>446</td>
<td>The target vector's extent is too big.</td>
</tr>
<tr>
<td>LAPI_ERR_TGT_OFFSET</td>
<td>496</td>
<td>Non-valid target offset.</td>
</tr>
<tr>
<td>LAPI_ERR_TGT_PURGED</td>
<td>421</td>
<td>The destination task is purged.</td>
</tr>
<tr>
<td>LAPI_ERR_TGT_STRIDE</td>
<td>447</td>
<td>The target vector's stride is less than its block.</td>
</tr>
<tr>
<td>LAPI_ERR_TGT_VAR_NULL</td>
<td>463</td>
<td>LAPI_Rmw's tgt_var address is NULL.</td>
</tr>
<tr>
<td>LAPI_ERR_TGT_VEC_ADDR</td>
<td>448</td>
<td>The target vector address is NULL, but its len is greater than 0.</td>
</tr>
<tr>
<td>LAPI_ERR_TGT_VEC_LEN</td>
<td>449</td>
<td>The target vector's length is too big.</td>
</tr>
<tr>
<td>LAPI_ERR_TGT_VEC_NULL</td>
<td>451</td>
<td>The target vector pointer is NULL.</td>
</tr>
<tr>
<td>LAPI_ERR_TGT_VEC_TYPE</td>
<td>452</td>
<td>The target vector type is not valid.</td>
</tr>
<tr>
<td>LAPI_ERR_TIMEOUT</td>
<td>413</td>
<td>A communication timeout has occurred. This error is returned in an asynchronous error handler.</td>
</tr>
<tr>
<td>LAPI_ERR_TRIGGER_UNKNOWN</td>
<td>522</td>
<td>The trigger is not registered.</td>
</tr>
<tr>
<td>LAPI_ERR_UDP_PKT_SZ</td>
<td>415</td>
<td>The UDP packet size is not valid.</td>
</tr>
<tr>
<td>LAPI_ERR_UDP_PORT_INFO</td>
<td>475</td>
<td>The udp_port information pointer is NULL.</td>
</tr>
<tr>
<td>LAPI_ERR_UDP_SOCKET</td>
<td>481</td>
<td>An error occurred during a UDP socket operation.</td>
</tr>
<tr>
<td>LAPI_ERR_UHDR_LEN</td>
<td>430</td>
<td>uhdr_len is too big.</td>
</tr>
<tr>
<td>LAPI_ERR_UHDR_NULL</td>
<td>429</td>
<td>uhdr is NULL, but uhdr_len is greater than 0.</td>
</tr>
<tr>
<td>Return value</td>
<td>Return code</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LAPI_ERR_UNKNOWN</td>
<td>400</td>
<td>An asynchronous, internal communication error occurred.</td>
</tr>
<tr>
<td>LAPI_ERR_USER_UDP_HNDLR_FAIL</td>
<td>416</td>
<td>The user-defined <em>udp_hndlr</em> failed.</td>
</tr>
<tr>
<td>LAPI_ERR_UTIL_CMD</td>
<td>477</td>
<td>The <em>LAPI_Util</em> command type is not valid.</td>
</tr>
<tr>
<td>LAPI_ERR_VEC_LEN_DIFF</td>
<td>455</td>
<td>The source and target vectors have different <em>len[]</em> values.</td>
</tr>
<tr>
<td>LAPI_ERR_VEC_NUM_DIFF</td>
<td>453</td>
<td>The source and target vectors have different <em>num_vecs</em> values.</td>
</tr>
<tr>
<td>LAPI_ERR_VEC_TYPE_DIFF</td>
<td>454</td>
<td>The source and target vectors have different <em>vec_type</em> values.</td>
</tr>
<tr>
<td>LAPI_ERR_XFER_CMD</td>
<td>476</td>
<td>The <em>LAPI_Xfer</em> command type is not valid.</td>
</tr>
<tr>
<td>LAPI_ERR_XLATE_FAILED</td>
<td>513</td>
<td>RDMA translation failed.</td>
</tr>
<tr>
<td>LAPI_SUCCESS</td>
<td>0</td>
<td>The LAPI function call completed successfully.</td>
</tr>
</tbody>
</table>
Appendix E. LAPI environment variables and runtime attributes

This appendix summarizes the LAPI environment variables and runtime attributes. For information about MPI and POE environment variables, see PE: Operation and Use, Volume 1.

Environment variables

Variables for communication

Table 61 includes LAPI environment variables for communication:

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Set:</th>
<th>Possible values</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MP_CLOCK_SOURCE</td>
<td></td>
<td>To use the InfiniBand interconnect clock as a time source (for AIX) or to specify the time source (for Linux). See the PE: MPI Programming Guide for more information.</td>
<td>For AIX: AIX SWITCH (HPS) For Linux: OS</td>
</tr>
<tr>
<td>MP_DEVTYPE</td>
<td></td>
<td>To specify the type of non-HPS user space device across which to perform communication.</td>
<td>ib (InfiniBand)</td>
</tr>
<tr>
<td>MP_MSG_API</td>
<td></td>
<td>To indicate to POE which parallel APIs (or protocols) are being used by the application.</td>
<td>See &quot;Setting environment variables&quot; on page 39</td>
</tr>
</tbody>
</table>

Variables for data transfer

Table 62 includes LAPI environment variables for data transfer:

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Set:</th>
<th>Possible values</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPI_VERIFY DGSP</td>
<td></td>
<td>To verify every DGSP at registration time. By default, this variable is set to no because it degrades performance. If it is set to yes, LAPI performs limited correctness checking of users' DGSPs at registration. It is recommended that applications in which users build DGSPs be tested with this variable set to yes, then run with it set to no during performance-critical operation. Note that many DGSP errors are only detectable during data transfer.</td>
<td>yes no</td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 2003, 2009
<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Set:</th>
<th>Possible values</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MP_BULK_MIN_MSG_SIZE</strong></td>
<td>To change the minimum message size (in bytes) for which LAPI will attempt to make bulk transfers. If you specify a value that is less than 4K, <strong>MP_BULK_MIN_MSG_SIZE</strong> is set to 4K. This environment variable is a hint that may or may not be honored by the communication library. See &quot;Bulk message transfer on AIX&quot; on page 63 for more information.</td>
<td>Any value greater than 4K</td>
<td>150K</td>
</tr>
</tbody>
</table>
| **MP_FIFO_MTU**               | To control the maximum transmission unit (MTU) size for FIFO mode. If the value is set to 4096, system administrators need to make sure that 4K MTU is enabled consistently on all of the InfiniBand switches in a cluster. Users need to get input from the system administrator about 4K MTU availability before enabling 4K MTU for their programs. Running with the 4K MTU option in protocols where it is not enabled in the switch can result in programs hanging. | 2048
4096                  | 2048          |
| **MP_RC_MAX_QP**             | To specify the maximum number of RC QPs that can be created.         | Any positive integer | 8156          |
| **MP_RC_USE_LMC**            | To enable or disable LID Mask Control (LMC). Enabling the use of LMC can improve performance, because a single port can support multiple reliable connected (RC) paths. A value of no (the default) indicates that only one RC connected path is supported. Setting **MP_RC_USE_LMC** to yes causes multiple RC paths to be supported, which may improve performance. | no
yes                  | no            |
| **MP_RCXT_BLKS**             | By POE. LAPI uses this value to determine the number of rCxt blocks that have been allocated for bulk transfer and user-initiated RDMA operations. | | |
Table 62. Environment variables for data transfer (continued)

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Set:</th>
<th>Possible values</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MP_RDMA_MTU</strong></td>
<td>Valid on Power 575 systems with the IBM GX Dual-Port 4x IB Host Channel Adapter</td>
<td>To control the MTU size for RDMA mode. If the value is set to <strong>4096</strong>, system administrators need to make sure that 4K MTU is enabled consistently on all of the InfiniBand switches in a cluster. Users need to get input from the system administrator about 4K MTU availability before enabling 4K MTU for their programs. Running with the 4K MTU option in protocols where it is not enabled in the switch can result in programs hanging.</td>
<td>2048 4096</td>
</tr>
<tr>
<td><strong>MP_RFIFO_SIZE</strong></td>
<td>To control the size, in bytes, of a receiving FIFO. If this environment variable is set, it overrides the default value.</td>
<td>any integer</td>
<td>4194304 (for fewer than 2048 tasks) 16777216 (for 2048 tasks or more)</td>
</tr>
<tr>
<td><strong>MP_USE_BULK_XFER</strong></td>
<td>Valid on RSCT LAPI for AIX systems running over US</td>
<td>To enable or disable bulk message transfer using the remote direct memory access (RDMA) protocol. This environment variable is a hint that may or may not be honored by the communication library. See &quot;Bulk message transfer on AIX&quot; on page 63 for more information.</td>
<td>yes no</td>
</tr>
</tbody>
</table>

**Variables for diagnostics**

Table 63 includes LAPI environment variables for diagnostics.

Table 63. Environment variables for diagnostics

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Set:</th>
<th>Possible values</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MP_DEBUG_NOTIMEOUT</strong></td>
<td>To attach to one or more tasks without the concern that some other task may reach the LAPI timeout. Such a timeout would normally occur if one of the job tasks was continuing to run and tried to communicate with a task to which the programmer has attached using a debugger. With this variable set, LAPI never times out and continues retransmitting message packets forever. The default setting (no) lets LAPI time out.</td>
<td>Any non-null string</td>
<td>no</td>
</tr>
</tbody>
</table>

**Variables for multicasting**

Table 64 on page 320 includes LAPI environment variables for multicasting.
### Table 64. Environment variables for multicasting

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Set:</th>
<th>Possible values</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MP_MC_INET_ADDR_LEN</strong></td>
<td>To determine the length of the available multicast address range, starting from the value specified for <strong>MP_MC_INET_BASE_ADDR</strong>.</td>
<td>any integer greater than or equal to 1 that does not exceed the range of multicast addresses that the operating system permits</td>
<td>16</td>
</tr>
<tr>
<td><strong>MP_MC_INET_BASE_ADDR</strong></td>
<td>To determine the base IPv4 multicast address for the current LAPI job.</td>
<td>any IPv4 multicast address</td>
<td>224.0.0.1</td>
</tr>
<tr>
<td><strong>MP_MC_INET_PORT</strong></td>
<td>To determine the port number used for multicasting.</td>
<td>any integer from 1024 through 65535</td>
<td>2008</td>
</tr>
<tr>
<td><strong>MP_USE_MC</strong></td>
<td>To determine whether to leverage the hardware multicasting function or use the peer-to-peer method to simulate multicasting. In IP mode, a value of <strong>yes</strong> indicates that the IP hardware multicasting function will be used and a value of <strong>no</strong> indicates that the peer-to-peer method will be used. In US mode, only the peer-to-peer method is supported.</td>
<td>no, yes</td>
<td>no</td>
</tr>
</tbody>
</table>

### Variables for performance tuning

Table 65 includes LAPI environment variables that are considered user-tunable for performance. See "Tunable environment variables" on page 107 for more information.

**Table 65. Environment variables for performance tuning**

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Set:</th>
<th>Possible values</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MP_ACK_THRESH</strong></td>
<td>The number of packets that are received before LAPI returns a batch of acknowledgments to the sending task.</td>
<td>A positive integer from 1 to 31</td>
<td>Depends on which communication adapter is used</td>
</tr>
<tr>
<td><strong>MP_POLLING_INTERVAL</strong></td>
<td>To control the interval for LAPI timer pops (in microseconds).</td>
<td>Any value greater than 10000</td>
<td>400000 (400 milliseconds)</td>
</tr>
<tr>
<td><strong>MP_RETRANSMIT_INTERVAL</strong></td>
<td>To control how often the communication subsystem library checks to see if it should retransmit packets that have not been acknowledged. The value specified is the number of polling loops between checks.</td>
<td>1000 to INT_MAX</td>
<td>1000000</td>
</tr>
</tbody>
</table>
### Table 65. Environment variables for performance tuning (continued)

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Set:</th>
<th>Possible values</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MP_REXMIT_BUF_CNT</strong></td>
<td>Specifies the number of buffers that LAPI must allocate. The size of each buffer is defined by <strong>MP_REXMIT_BUF_SIZE</strong>. This count indicates the number of in-flight messages smaller than <strong>MP_REXMIT_BUF_SIZE</strong> that LAPI can store in its local buffers in order to free up the user’s message buffers more quickly.</td>
<td>Any integer greater than 0</td>
<td>128</td>
</tr>
<tr>
<td><strong>MP_REXMIT_BUF_SIZE</strong></td>
<td>The maximum message size, in bytes, that LAPI will store in its local buffers in order to more quickly free up the user buffer containing message data. This size indicates the size of the local buffers LAPI will allocate to store such messages, and will impact memory usage, while potentially improving performance. LAPI will use the buffer to store the user header and the user data.</td>
<td>Any integer greater than 0</td>
<td>16384</td>
</tr>
<tr>
<td><strong>MP_UDP_PACKET_SIZE</strong></td>
<td>To control the size of the LAPI packets that are used for UDP data transfer. LAPI initialization will fail if this environment variable is set to a value that is outside the valid range.</td>
<td>1024 to 65536</td>
<td>8192 for non-switch devices 65536 for switch devices</td>
</tr>
</tbody>
</table>

### Variables for POE

Table 66 includes LAPI environment variables for POE. The variables described in this section are set by the user and interpreted by POE. Although LAPI ignores these variables (except **MP_INFOLEVEL**), they are included here because they have an impact on LAPI jobs.
Table 66. Environment variables for POE

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Set:</th>
<th>Possible values</th>
<th>Default value</th>
</tr>
</thead>
</table>
| **MP_EUIDevice**     | To specify the type of network adapter that will be used for message passing. | For RSCT LAPI for AIX: 
  - *adapter_device_name* as configured in LoadLeveler 
  - *en* Ethernet 
  - *fi0* FDDI 
  - *ml0* IP multi-link device 
  - *network_type* as configured in LoadLeveler 
  - *sn_all* one or more windows per network 
  - *sn_single* all windows are on a single network 
  - *tr0* token ring 
  For LAPI for Linux: 
  - *adapter_device_name* as configured in LoadLeveler 
  - *ethn* Gigabit Ethernet 
  - *network_type* as configured in LoadLeveler 
  - *sn_all* one or more windows per network 
  - *sn_single* all windows are on a single network | The adapter type that is used as the external network address. 
  - For IP: **en0** 
  - For US on LAPI for Linux or RSCT LAPI for AIX 5.3: **sn_single** |
| **MP_EUILIB**        | The communication subsystem library implementation to use for communication: either the Internet Protocol (IP) communication subsystem or the user space (US) communication subsystem. | These values are case-sensitive: 
  - **ip** 
  - **us** | **ip** |
Table 66. Environment variables for POE (continued)

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Set:</th>
<th>Possible values</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MP_INFOLEVEL</strong></td>
<td>The level of message reporting. Any value that is greater than or equal to 2 causes LAPI to print out such library information as the LAPI version number and the build timestamp. Levels 3, 4, 5, and 6 also include high-level diagnostic messages for use by the IBM Support Center.</td>
<td>One of the following integers: 0 Error. 1 Warning and error. 2 Informational, warning, and error. 3 Informational, warning, error, and job-level debugging. 4 Informational, warning, error, job-level debugging, and node-level debugging. 5 Informational, warning, error, job-level debugging, node-level debugging, and task-level debugging. 6 Informational, warning, error, job-level debugging, node-level debugging, task-level debugging, and event-level debugging.</td>
<td>0</td>
</tr>
<tr>
<td><strong>MP_INSTANCES</strong></td>
<td>To control the number of instances of IP addresses or user space windows to be assigned.</td>
<td>a positive integer or the string <strong>max</strong></td>
<td>1</td>
</tr>
</tbody>
</table>

Variables for shared memory

Table 67 includes LAPI environment variables for shared memory jobs. Set these variables if you are using LAPI on a standalone system.

Table 67. Environment variables for shared memory

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Set:</th>
<th>Possible values</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LAPI_USE_SHM</strong></td>
<td>To enable or disable the use of shared memory. <strong>no</strong> -- disables the use of shared memory (the default). <strong>yes</strong> -- enables the use of shared memory where it is possible. LAPI will communicate using shared memory among all common tasks (tasks that are on the same node) over the selected device (user space over switch, IP over switch, or IP over Ethernet). See <strong>MP_EUIDEVICE</strong> and <strong>MP_EUILIB</strong> for tasks on different nodes. Shared memory requires segment registers, which can affect availability to user code in 32-bit applications. <strong>only</strong> -- communicates only using shared memory. LAPI will fail to initialize if this option is chosen and tasks are assigned to more than one node.</td>
<td><strong>yes</strong> no only</td>
<td><strong>no</strong></td>
</tr>
</tbody>
</table>
**Variables for standalone systems**

When LAPI is running in a PE environment, POE sets some of the variables that are described in Table 68 (MP_CHILD, MP_COMMON_TASKS, MP_PARTITION, and MP_PROCS).

When LAPI is running in standalone mode, you need to set the variables that are described in Table 68 explicitly. See “Setting up a standalone system” on page 141 for more information.

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Set:</th>
<th>Possible values</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MP_CHILD</td>
<td>The task ID of the current job. MP_CHILD needs to be set to a unique value for each task in standalone mode.</td>
<td>Any value that is greater than or equal to 0 and less than the value of MP_PROCS</td>
<td>you need to set</td>
</tr>
<tr>
<td>MP_COMMON_TASKS</td>
<td>For shared memory jobs.</td>
<td>See “Setting up a standalone system” on page 141</td>
<td>you need to set</td>
</tr>
<tr>
<td>MP_LAPI_INET_ADDR</td>
<td>The network setup among LAPI tasks for IP communication.</td>
<td>See “Setting up a standalone system” on page 141</td>
<td>you need to set</td>
</tr>
<tr>
<td>MP_LAPI_NETWORK</td>
<td>LAPI network information.</td>
<td>See “Setting up a standalone system” on page 141</td>
<td>you need to set</td>
</tr>
<tr>
<td>MP_PARTITION</td>
<td>A number that is the same for all tasks in the job. In standalone mode, you need to set this variable to an identical value for each task. In standalone mode for switched communication, the MP_PARTITION value must be associated with the network table description file.</td>
<td>any value</td>
<td>you need to set</td>
</tr>
<tr>
<td>MP_PROCS</td>
<td>The value of num_tasks, which is the total number of program tasks in the job. This number must be the same for all tasks.</td>
<td>A positive integer from 1 to the maximum number of tasks that is supported by the configuration.</td>
<td>you need to set</td>
</tr>
</tbody>
</table>

The descriptions and formats of MP_COMMON_TASKS, MP_LAPI_INET_ADDR, MP_LAPI_NETWORK are provided in this book for informational purposes only. These environment variables are not intended to be used as external programming interfaces. IBM will not guarantee that the formats or values of these variables can continue to be used without change in future releases. Programmers and users who choose to develop applications that depend on these variables do so with the understanding that these variables may be subject to future change. IBM cannot guarantee that such applications can migrate or coexist with future releases without additional changes, nor will IBM ensure that there will be binary compatibility of these variables.
### Runtime attributes

This section includes attributes that you can query during runtime using the `LAPI_Qenv` interface. See "LAPI_Qenv" on page 221 for more information about using `LAPI_Qenv`.

You can set some of these attributes using `LAPI_Senv`. See "Attributes you can query or set." For more information about using `LAPI_Senv`, see "LAPI_Senv" on page 234.

### Attributes you can query or set

Table 69 includes runtime attributes you can set using `LAPI_Senv` or query using `LAPI_Qenv`.

**Table 69. Runtime attributes you can query or set**

<table>
<thead>
<tr>
<th>Runtime attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACK_THRESHOLD</td>
<td>This value represents the number of packets received before LAPI sends acknowledgements. It can also be set using <code>LAPI_Senv</code> or with the <code>MP_ACK_THRESH</code> environment variable (see &quot;Variables for performance tuning&quot; on page 320).</td>
</tr>
<tr>
<td>ERROR_CHK</td>
<td>This attribute is a user-settable toggle that indicates whether LAPI should perform error checking. If set, LAPI calls will perform bounds-checking on parameters. Due to the potential performance degradation, error checking is disabled by default.</td>
</tr>
<tr>
<td>INTERRUPT_SET</td>
<td>This a user-settable toggle value that controls whether LAPI runs with interrupts turned on or off. With interrupts on, a timer-driven interrupt will drive packet acknowledgements and retransmits.</td>
</tr>
<tr>
<td>TIMEOUT</td>
<td>This value corresponds to number of seconds that LAPI should wait on receiving packet acknowledgements before considering a remote task as unreachable. It can be set to a value in the range MIN_TIMEOUT &lt; TIMEOUT &lt; MAX_TIMEOUT. The default is 900 seconds (15 minutes).</td>
</tr>
</tbody>
</table>

### Attributes you can query

This section includes attribute values that you can query during runtime using the `LAPI_Qenv` interface. `LAPI_Qenv` returns values through a reference parameter.

### Attributes that return integers

Table 70 includes attributes that return integers. The actual parameter is expected to be of type `&int`.

**Table 70. Attributes that return integers**

<table>
<thead>
<tr>
<th>Runtime attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUF_CP_SIZE</td>
<td>This represents the value of LAPI’s send-side copy buffer. It can be set at job startup using the <code>MP_REXMIT_BUF_SIZE</code> environment variable (see &quot;Variables for performance tuning&quot; on page 320).</td>
</tr>
<tr>
<td>BULK_MIN_MSG_SIZE</td>
<td>This represents the current minimum message size that will be used for bulk transfer. Valid on RSCT LAPI for AIX systems running over US.</td>
</tr>
<tr>
<td>BULK_XFER</td>
<td>This value indicates whether bulk transfer is enabled (1) or disabled (0). Valid on RSCT LAPI for AIX systems running over US.</td>
</tr>
<tr>
<td>LOC_ADDRTBL_SZ</td>
<td>This value represents the upper bound on LAPI’s internal table size. For example, the size of the address table used in <code>LAPI.Addr.set</code> is bounded by this value.</td>
</tr>
<tr>
<td>MAX_ATOM_SIZE</td>
<td>This represents the maximum atom size for user DGSPs. See &quot;Using data gather/scatter programs (DGSPs)&quot; on page 52 for more information.</td>
</tr>
</tbody>
</table>
Table 70. Attributes that return integers (continued)

<table>
<thead>
<tr>
<th>Runtime attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX_DATA_SZ</td>
<td>This query is deprecated. For 32-bit applications, it will return the value of the macro LAPI_MAX_MSG_SZ defined in lapi.h. For 64-bit applications, it will return the largest unsigned integer. Rather than use this query, it is recommended that users use the macro LAPI_MAX_MSG_SZ directly, which is valid for both 32-bit and 64-bit applications.</td>
</tr>
<tr>
<td>MAX_PKT_SZ</td>
<td>This value represents the maximum storage for user data (header + data) in each LAPI packet.</td>
</tr>
<tr>
<td>MAX_PKTS_OUT</td>
<td>This value represents the maximum number of packets that can be “in flight” between any two tasks.</td>
</tr>
<tr>
<td>MAX_PORTS</td>
<td>This value represents the maximum number of LAPI instances that are available for use.</td>
</tr>
<tr>
<td>MAX_TIMEOUT</td>
<td>This is the maximum number of seconds that a user can set for TIMEOUT. If an attempt is made to set TIMEOUT to a value outside the valid range, LAPI_Senv will return an error and TIMEOUT will be unchanged.</td>
</tr>
<tr>
<td>MAX_UHDR_SZ</td>
<td>The maximum size in bytes that can be used for user header data. See <a href="#">LAPI_Amsend on page 168</a> for more information on using a user header.</td>
</tr>
<tr>
<td>MIN_TIMEOUT</td>
<td>This is the minimum number of seconds that a user can set for TIMEOUT. If an attempt is made to set TIMEOUT to a value outside the valid range, LAPI_Senv will return an error and TIMEOUT will be unchanged.</td>
</tr>
<tr>
<td>NUM_REX_BUFS</td>
<td>This value represents the number of retransmission buffers that LAPI uses.</td>
</tr>
<tr>
<td>NUM_TASKS</td>
<td>The total number of tasks in the job. This corresponds to the value set in MP_PROCS.</td>
</tr>
<tr>
<td>QUERY_SHM_ENABLED</td>
<td>This is a boolean value that indicates whether LAPI is communicating using shared memory.</td>
</tr>
<tr>
<td>QUERY_SHM_NUM_TASKS</td>
<td>This value represents the number of tasks with which the current task can communicate using shared memory.</td>
</tr>
<tr>
<td>RDMA_REMOTE_RCXT_AVAIL</td>
<td>Contains a pointer to an integer that stores the number of locally-available rCxts that can be used as remote rCxts on other tasks.</td>
</tr>
<tr>
<td>RDMA_REMOTE_RCXT_TOTAL</td>
<td>Contains a pointer to an integer that stores the total number of local rCxts that are remote rCxts or that can be used as remote rCxts on other tasks.</td>
</tr>
<tr>
<td>REX_BUF_SZ</td>
<td>This value represents the size of LAPI’s retransmission buffers.</td>
</tr>
<tr>
<td>TASK_ID</td>
<td>The ID that LAPI has for the given task. For standalone jobs, it corresponds to the value set in MP_CHILD.</td>
</tr>
</tbody>
</table>

Attributes that return multiple values

Table 71 includes runtime attributes that return multiple values. These attributes expect a pointer to a different datatype and require that the actual parameter represents sufficient space for the query type. LAPI casts the passed pointer to this type.

Table 71. Attributes that return multiple values

<table>
<thead>
<tr>
<th>Runtime attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRINT_STATISTICS</td>
<td>Calling LAPI_Qenv with this query value will cause LAPI to dump the values that would be returned in a lapi_statistics_t structure to standard output.</td>
</tr>
</tbody>
</table>
### Table 71. Attributes that return multiple values (continued)

<table>
<thead>
<tr>
<th>Runtime attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>QUERY_LOCAL_SEND_STATISTICS</strong></td>
<td>Returns a number of statistics about the running environment, for the local copy path. Expects a pointer to type <code>lapi_statistics_t</code>. For example:</td>
</tr>
</tbody>
</table>
|                                            | ```
|                                            | { lapi_statistics_t stats;
|                                            | LAPI_Qenv(handle, QUERY_LOCAL_SEND_STATISTICS, (int *)&stats));
|                                            | }                                                                                                                                 |
|                                            | Note that the address of `stats` is cast to `int *`. This is required to match the signature of `LAPI_Qenv`.                                   |
| **QUERY_SHM_STATISTICS**                   | Returns a number of statistics about the running environment, for the shared memory path. Expects a pointer to type `lapi_statistics_t`. For example: |
|                                            | ```
|                                            | { lapi_statistics_t stats;
|                                            | LAPI_Qenv(handle, QUERY_SHM_STATISTICS, (int *)&stats));
|                                            | }                                                                                                                                 |
|                                            | Note that the address of `stats` is cast to `int *`. This is required to match the signature of `LAPI_Qenv`.                                   |
| **QUERY_SHM_TASKS**                        | This query returns a list of shared memory task IDs for each task with which this task can communicate using shared memory. LAPI expects an array large enough to hold an integer index for each task in the job. For example: |
|                                            | ```
|                                            | { int *shm_task_list;
|                                            | int i;
|                                            | LAPI_Qenv(hndl, NUM_TASKS, &num_tasks);
|                                            | shm_task_list = (int *) (malloc(sizeof(int)*num_tasks));
|                                            | LAPI_Qenv(hndl, QUERY_SHM_TASKS, shm_task_list);
|                                            | for( i = 0; i < num_tasks; i++ ) {
|                                            |     printf("task[%d] has shm_task_id %d, (num_tasks:%d)\n", i, shm_task_list[i], num_tasks);
|                                            | }
|                                            | free(shm_task_list);                                                                                                                        |
| **QUERY_STATISTICS**                       | Returns a number of statistics about the running environment. Expects a pointer to type `lapi_statistics_t`. For example: |
|                                            | ```
|                                            | { lapi_statistics_t stats;
|                                            | LAPI_Qenv(handle, QUERY_STATISTICS, (int *)&stats));
|                                            | }                                                                                                                                 |
|                                            | Note that the address of `stats` is cast to `int *`. This is required to match the signature of `LAPI_Qenv`.                                   |

### Attributes for legacy code

LAPI no longer uses the following runtime attributes, which are included for support of legacy code: `EPOCH_NUM`, `RCV_FIFO_SIZE`, and `USE_THRESH`. 
# Appendix F. LAPI datatypes

This appendix lists datatypes that you can use in your LAPI programs.

<table>
<thead>
<tr>
<th>C datatype</th>
<th>FORTRAN datatype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>com_thread_info_t</td>
<td>com_thread_info_t</td>
<td>For thread attribute and initialization functions.</td>
</tr>
<tr>
<td>compl_hndlr_t</td>
<td>compl_hndlr_t</td>
<td>The receive completion handler.</td>
</tr>
<tr>
<td>ddm_func_t</td>
<td>ddm_func_t</td>
<td>For data distribution manager (DDM) functions.</td>
</tr>
<tr>
<td>hdr_hndlr_t</td>
<td>hdr_hndlr_t</td>
<td>The header handler for a contiguous DGSP message.</td>
</tr>
<tr>
<td>in_addr_t</td>
<td>in_addr_t</td>
<td>For local IP addresses.</td>
</tr>
<tr>
<td>in_port_t</td>
<td>in_port_t</td>
<td>For local port addresses.</td>
</tr>
<tr>
<td>lapi_add_udp_port_ext</td>
<td>No equivalent type</td>
<td>The <code>LAPI_Util</code> command structure (in C) for updating the UDP port information of the destination task.</td>
</tr>
<tr>
<td>lapi_add_udp_port_t</td>
<td>lapi_add_udp_port_t</td>
<td>The datatype (in FORTRAN) for updating the UDP port information of the destination task. (deprecated in C)</td>
</tr>
<tr>
<td>(C equivalent: void *)</td>
<td>lapi_addr_type</td>
<td>For address functions.</td>
</tr>
<tr>
<td>lapi_am_t</td>
<td>lapi_am_t</td>
<td>The <code>LAPI_Xfer</code> command structure (in C) or the datatype (in FORTRAN) for one contiguous active message.</td>
</tr>
<tr>
<td>lapi_amdgsp_t</td>
<td>lapi_amdgsp_t</td>
<td>The <code>LAPI_Xfer</code> command structure (in C) or the datatype (in FORTRAN) for one DGSP active message.</td>
</tr>
<tr>
<td>lapi_amv_t</td>
<td>lapi_amv_t</td>
<td>The <code>LAPI_Xfer</code> command structure (in C) or the datatype (in FORTRAN) for one vector active message.</td>
</tr>
<tr>
<td>lapi_cntr_t</td>
<td>lapi_cntr_t</td>
<td>Defines a LAPI counter.</td>
</tr>
<tr>
<td>lapi_cond_destroy_t</td>
<td>lapi_cond_destroy_t</td>
<td>Destroy condition function pointer.</td>
</tr>
<tr>
<td>lapi_cond_init_t</td>
<td>lapi_cond_init_t</td>
<td>Initialize condition function pointer.</td>
</tr>
<tr>
<td>lapi_cond_signal_t</td>
<td>lapi_cond_signal_t</td>
<td>Condition signal function pointer.</td>
</tr>
<tr>
<td>lapi_cond_timedwait_t</td>
<td>lapi_cond_timedwait_t</td>
<td>Condition timed wait function pointer.</td>
</tr>
<tr>
<td>lapi_cond_wait_t</td>
<td>lapi_cond_wait_t</td>
<td>Condition wait function pointer.</td>
</tr>
<tr>
<td>lapi_ctl_flags_t</td>
<td>No equivalent type</td>
<td>In C: part of the <code>lapi_return_info_t</code> structure. Instructs LAPI on what it should do with a message after the header handler is called.</td>
</tr>
<tr>
<td>lapi_dev_t</td>
<td>lapi_dev_t</td>
<td>For protocol devices.</td>
</tr>
<tr>
<td>lapi dg_handle_t</td>
<td>lapi dg_handle_t</td>
<td>Defines a DGSP handle.</td>
</tr>
<tr>
<td>lapi dgsm_block_t</td>
<td>lapi dgsm_block_t</td>
<td>Defines a DGSP BLOCK instruction.</td>
</tr>
<tr>
<td>lapi dgsm_control_t</td>
<td>lapi dgsm_control_t</td>
<td>Defines a DGSP CONTROL instruction.</td>
</tr>
<tr>
<td>lapi dgsm_copy_t</td>
<td>lapi dgsm_copy_t</td>
<td>Defines a DGSP COPY instruction.</td>
</tr>
<tr>
<td>lapi dgsm_gosub_t</td>
<td>lapi dgsm_gosub_t</td>
<td>Defines a DGSP GOSUB instruction.</td>
</tr>
<tr>
<td>lapi dgsm_iterate_t</td>
<td>lapi dgsm_iterate_t</td>
<td>Defines a DGSP ITERATE instruction.</td>
</tr>
<tr>
<td>C datatype</td>
<td>FORTRAN datatype</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>lapi_dgsm_mcopy_t</td>
<td>lapi_dgsm_mcopy_t</td>
<td>Defines a DGSP multiple copy (MCOPY) instruction.</td>
</tr>
<tr>
<td>lapi_dgsp_density_t</td>
<td>lapi_dgsm_contig</td>
<td>In C: an enumeration for values in a LAPI DGSP descriptor.</td>
</tr>
<tr>
<td></td>
<td>lapi_dgsm_sparse</td>
<td>In FORTRAN: one of three DGSP data layout types.</td>
</tr>
<tr>
<td></td>
<td>lapi_dgsm_unit</td>
<td></td>
</tr>
<tr>
<td>lapi_dgsp_descr_t</td>
<td>lapi_dgsp_descr_t</td>
<td>A DGSP descriptor structure.</td>
</tr>
<tr>
<td>lapi_dgsp_handle_t</td>
<td>lapi_dgsp_handle_t</td>
<td>A handle for a registered DGSP.</td>
</tr>
<tr>
<td>lapi_dref_dgsp_t</td>
<td>lapi_dref_dgsp_t</td>
<td>The LAPI_Util command structure (in C) or the datatype (in FORTRAN) for a DGSP un-reserve operation.</td>
</tr>
<tr>
<td>lapi_err_t</td>
<td>lapi_err_t</td>
<td>Error type.</td>
</tr>
<tr>
<td>lapi_extend_t</td>
<td>lapi_extend_t</td>
<td>Additional structure extension.</td>
</tr>
<tr>
<td>lapi_get_t</td>
<td>lapi_get_t</td>
<td>The LAPI_Xfer command structure (in C) or the datatype (in FORTRAN) for one “get” message.</td>
</tr>
<tr>
<td>lapi_get_pvo_t</td>
<td>lapi_get_pvo_t</td>
<td>The LAPI_Util command structure (in C) or the datatype (in FORTRAN) that is used to register or deregister a memory region.</td>
</tr>
<tr>
<td>lapi_getv_t</td>
<td>lapi_getv_t</td>
<td>The LAPI_Xfer command structure (in C) or the datatype (in FORTRAN) for one vector “get” message.</td>
</tr>
<tr>
<td>lapi_group_t</td>
<td>lapi_group_t</td>
<td>Group handle used in LAPI calls.</td>
</tr>
<tr>
<td>lapi_handle_t</td>
<td>lapi_handle_t</td>
<td>An opaque handle for identifying the LAPI context.</td>
</tr>
<tr>
<td>lapi_hwxfer_t</td>
<td>lapi_hwxfer_t</td>
<td>The LAPI_Xfer command structure (in C) or the datatype (in FORTRAN) for an RDMA transfer.</td>
</tr>
<tr>
<td>lapi_info_t</td>
<td>lapi_info_t</td>
<td>Command structure for LAPI_Init.</td>
</tr>
<tr>
<td>lapi_lib_t</td>
<td>lapi_lib_t</td>
<td>For the LAPI library version.</td>
</tr>
<tr>
<td>lapi_long_t</td>
<td>lapi_long_type</td>
<td>LAPI long type.</td>
</tr>
<tr>
<td>long long</td>
<td>lapi_long_long_type</td>
<td>LAPI long long type.</td>
</tr>
<tr>
<td>lapi_lvec_t</td>
<td>lapi_lvec_t</td>
<td>LAPI long vector type.</td>
</tr>
<tr>
<td>lapi_mc_t</td>
<td>lapi_mc_t</td>
<td>The LAPI_Xfer command structure (in C) or the datatype (in FORTRAN) for multicasting.</td>
</tr>
<tr>
<td>lapi_msg_info_t</td>
<td>lapi_msg_info_t</td>
<td>Information about a LAPI_Msgpoll call.</td>
</tr>
<tr>
<td>lapi_msg_state_t</td>
<td>lapi_msg_state_t</td>
<td>In lapi_msg_info_t indicates whether there is any completion of send/receive after LAPI_Msgpoll returns.</td>
</tr>
<tr>
<td>lapi_msglen_t</td>
<td>lapi_msglen_t</td>
<td>LAPI message length.</td>
</tr>
<tr>
<td>lapi_mutex_getowner_t</td>
<td>lapi_mutex_getowner_t</td>
<td>Mutex get owner function pointer.</td>
</tr>
<tr>
<td>lapi_mutex_lock_t</td>
<td>lapi_mutex_lock_t</td>
<td>Mutex lock function pointer.</td>
</tr>
<tr>
<td>lapi_mutex_trylock_t</td>
<td>lapi_mutex_trylock_t</td>
<td>Mutex try lock function pointer.</td>
</tr>
<tr>
<td>lapi_mutex_unlock_t</td>
<td>lapi_mutex_unlock_t</td>
<td>Mutex unlock function pointer.</td>
</tr>
<tr>
<td>C datatype</td>
<td>FORTRAN datatype</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------</td>
<td>--------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>lapi_pack_dgsp_t</td>
<td>lapi_pack_dgsp_t</td>
<td>The <strong>LAPI_Util</strong> command structure (in C) or the datatype (in FORTRAN) for packing data from a memory layout defined by a DGSP to a contiguous buffer.</td>
</tr>
<tr>
<td>lapi_put_t</td>
<td>lapi_put_t</td>
<td>The <strong>LAPI_Xfer</strong> command structure (in C) or the datatype (in FORTRAN) for one “put” message.</td>
</tr>
<tr>
<td>lapi_putv_t</td>
<td>lapi_putv_t</td>
<td>The <strong>LAPI_Xfer</strong> command structure (in C) or the datatype (in FORTRAN) for one vector “put” message.</td>
</tr>
<tr>
<td>lapi_query_t</td>
<td></td>
<td>No equivalent type (query types are defined explicitly in the 32-bit and 64-bit versions of lapif.h). In C: an enumeration that defines all queries supported by <strong>LAPI_Qenv</strong> and <strong>LAPI_Senv</strong>.</td>
</tr>
<tr>
<td>lapi_rdma_flg_t</td>
<td>lapi_rdma_flg_t</td>
<td>Bit flags that are used to specify the type of target notification in an RDMA operation.</td>
</tr>
<tr>
<td>lapi_rdma_notification_t</td>
<td>lapi_rdma_notification_t</td>
<td>The <strong>LAPI_Util</strong> command structure (in C) or the datatype (in FORTRAN) for notification for a remote RDMA operation.</td>
</tr>
<tr>
<td>lapi_rdma_op_t</td>
<td>lapi_rdma_op_t</td>
<td>Bit flags that are used to specify the operation type (“get” or “put”) in an RDMA transfer.</td>
</tr>
<tr>
<td>lapi_rdma_req_t</td>
<td>lapi_rdma_req_t</td>
<td>Bit flags that are used for obtaining or releasing rCxts or for registering or deregistering RDMA memory.</td>
</tr>
<tr>
<td>lapi_rdma_tag_t</td>
<td>lapi_rdma_tag_t</td>
<td>An unsigned short value used for registering counters or callbacks that are used for requesting target notification on completion of an RDMA operation.</td>
</tr>
<tr>
<td>lapi_reg_ddm_t</td>
<td>lapi_reg_ddm_t</td>
<td>The <strong>LAPI_Util</strong> command structure (in C) or the datatype (in FORTRAN) for data distribution manager (DDM) functions.</td>
</tr>
<tr>
<td>lapi_reg_dgsp_t</td>
<td>lapi_reg_dgsp_t</td>
<td>The <strong>LAPI_Util</strong> command structure (in C) or the datatype (in FORTRAN) for registering a DGSP.</td>
</tr>
<tr>
<td>lapi_remote_cxt_t</td>
<td>lapi_remote_cxt_t</td>
<td>The <strong>LAPI_Util</strong> command structure (in C) or the datatype (in FORTRAN) for notification for a remote RDMA operation.</td>
</tr>
<tr>
<td>lapi_resv_dgsp_t</td>
<td>lapi_resv_dgsp_t</td>
<td>The <strong>LAPI_Util</strong> command structure (in C) or the datatype (in FORTRAN) for reserving a DGSP.</td>
</tr>
<tr>
<td>lapi_ret_flags_t</td>
<td></td>
<td>No equivalent type In C: part of the <strong>lapi_return_info_t</strong> structure. Indicates to LAPI whether to run the completion handler inline.</td>
</tr>
<tr>
<td>lapi_return_info_t</td>
<td></td>
<td>No equivalent type In C: a structure that extends the header handler interface to pass information between LAPI and a user program.</td>
</tr>
<tr>
<td>lapi_rmw_t</td>
<td>lapi_rmw_t</td>
<td>The <strong>LAPI_Xfer</strong> command structure (in C) or the datatype (in FORTRAN) for one read-modify-write operation.</td>
</tr>
<tr>
<td>lapi_sh_info_t</td>
<td>lapi_sh_info_t</td>
<td>Send completion handler information.</td>
</tr>
<tr>
<td><strong>C datatype</strong></td>
<td><strong>FORTRAN datatype</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>lapi_statistics_t</td>
<td>lapi_statistics_t</td>
<td>LAPI statistics.</td>
</tr>
<tr>
<td>lapi_trigger_function_t</td>
<td>lapi_trigger_function_t</td>
<td>Trigger function pointer.</td>
</tr>
<tr>
<td>lapi_trigger_util_t</td>
<td>lapi_trigger_util_t</td>
<td>Registers or deregisters a trigger function.</td>
</tr>
<tr>
<td>lapi_udpinfo_t</td>
<td>lapi_udpinfo_t</td>
<td>UDP information.</td>
</tr>
<tr>
<td>lapi_unpack_dgsp_t</td>
<td>lapi_unpack_dgsp_t</td>
<td>The <code>LAPI_Util</code> command structure (in C) or the datatype (in FORTRAN) for unpacking data from a contiguous buffer to a memory layout that is defined by a DGSP.</td>
</tr>
<tr>
<td>lapi_user_cxt_t</td>
<td>lapi_user_cxt_t</td>
<td>Specifies the context on the remote side.</td>
</tr>
<tr>
<td>lapi_user_pvo_t</td>
<td>lapi_user_pvo_t</td>
<td>Defines the user PVO.</td>
</tr>
<tr>
<td>lapi_usr_fcall_t</td>
<td>lapi_usr_fcall_t</td>
<td>For debugging only.</td>
</tr>
<tr>
<td>lapi_util_t</td>
<td>LAPI_ADD_UDP_DEST_PORT LAPI_DGSP_PACK LAPI_DGSP_UNPACK LAPI_GET_THREAD_FUNC LAPI_REG_DDM_FUNC LAPI_REGISTER_DGSP LAPI_REGISTER_NOTIFICATION LAPI_REMOTE_RCXT LAPI_reserve DGSP LAPI_unregister DGSP LAPI_XLATE_ADDRESS</td>
<td>In C: the union of all possible command structures for <code>LAPI_Util</code>. In FORTRAN: one of 11 types for <code>LAPI_Util</code>.</td>
</tr>
<tr>
<td>lapi_util_type_t</td>
<td>No equivalent type</td>
<td>In C: specifies the type of utility in the command structures for <code>LAPI_Util</code>.</td>
</tr>
<tr>
<td>lapi_vec_t</td>
<td>lapi_vec_t</td>
<td>Defines LAPI vector data layout.</td>
</tr>
<tr>
<td>lapi_vectype_t</td>
<td>lapi_vectype_t</td>
<td>The type of a vector.</td>
</tr>
<tr>
<td>lapi_xfer_t</td>
<td>LAPI_AM_XFER LAPI_AM_LW_XFER LAPI_AMV_XFER LAPI_AMX_XFER LAPI_DGSP_XFER LAPI_GET_XFER LAPI_GETV_XFER LAPI_MC_XFER LAPI_PUT_XFER LAPI_PUTV_XFER LAPI_RDMA_XFER LAPI_RMW_XFER</td>
<td>In C: the union of all possible command structures for <code>LAPI_Xfer</code>. In FORTRAN: one of 12 transfer types for <code>LAPI_Xfer</code>.</td>
</tr>
<tr>
<td>lapi_xfer_type_t</td>
<td>No equivalent type</td>
<td>In C: the type of a <code>LAPI_Xfer</code> command structure.</td>
</tr>
<tr>
<td>remote_handlr_t</td>
<td>remote_handlr_t</td>
<td>Defines a callback for an RDMA operation when target notification is required.</td>
</tr>
<tr>
<td>RMW_ops_t</td>
<td>RMW_ops_t</td>
<td>The type of read-modify-write operation.</td>
</tr>
<tr>
<td>scompl_hndlr_t</td>
<td>scompl_hndlr_t</td>
<td>The send completion handler.</td>
</tr>
<tr>
<td>vhdr_hndlr_t</td>
<td>vhdr_hndlr_t</td>
<td>The header handler for a vector message.</td>
</tr>
</tbody>
</table>
Appendix G. LAPI constants and size limits

This appendix includes information about LAPI constants and size limits.

The upper bound on the message size that a user can transfer (LAPI_MAX_MSG_SZ):
- For 32-bit applications: \( 0x7fffffff \)
- For 64-bit applications: \( 0x7fffffffffffffffLL \)

The maximum error string length (LAPI_MAX_ERR_STRING): 160 characters

The maximum number of nodes: 2048 for AIX, 1024 for Linux

The maximum number of tasks that are allowed within a single parallel job: 16384 for AIX, 2048 for Linux

The maximum number of shared memory tasks: 128
Appendix H. LAPI restrictions

General restrictions

Nodes within the same LAPI job must be at the same version of LAPI. Interoperability with previous versions of LAPI is not supported.

You cannot make LAPI calls from within the header handler. For contiguous data, you can copy the data to the buffer you specified. For non-contiguous data, you must pass the DGSP handle and a buffer address to LAPI. LAPI will unpack the data to the specified buffer address.

I/O operations and blocking calls, including blocking LAPI calls, should not be performed within an inline completion handler. Inline completion handlers should be short, because no progress can be made while the main thread is executing the handler. You must use caution with inline completion handlers so that LAPI's internal queues do not fill up while waiting for the handler to complete. Note that LAPI places no restrictions on completion handlers that are run "normally" (that is, by the completion handler thread).

User application threads and completion handler threads cannot hold mutual exclusion resources before making LAPI calls. If they do, it is possible to run into deadlock situations.

Static linking is not supported.

For systems running PE, both US and IP are supported for shared handles as long as they are the same for both handles. Mixed transport protocols such as LAPI IP and LAPI user space (US) are not supported.

Use of segment registers on AIX

For AIX, the user space (US) library uses two segment registers of the 10 that are unassigned in the user's AIX process space. Thus, the user can allocate a maximum of 8 segments (-bmaxdata=0x80000000) to extended heap for large data structures. However, programs compiled with eight segments may experience reduced shared memory performance in a 32-bit environment because there is no extra segment for LAPI to do address space attach, which avoids one extra copy for large messages. See "The shared memory execution model" on page 305 for more information.

Other restrictions

A maximum of 16MB of unaligned blocks of virtual memory (or 32MB aligned on a 16MB boundary) can be used in a call to LAPI_Util with the LAPI_XLATE_ADDRESS operation.

LAPI_Xfer with transfer type LAPI_RDMA_XFER is not supported for persistent LAPI jobs.

For restrictions related to:

- Low latency communication, see "Optimizing communication for small messages" on page 98
- Lock sharing, see "Implications and restrictions" on page 121
• Striping, failover, and recovery, see "Failover and recovery restrictions" on page 131 and "Communication and memory considerations for AIX" on page 133.
• Shared memory, see "Requirements and considerations" on page 307.
Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user’s responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.
Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

For AIX:
IBM Corporation
Department LRAS, Building 003
11400 Burnet Road
Austin, Texas 78758-3498
U.S.A.

For Linux:
IBM Corporation
Department LJEB, MS P905
2455 South Road
Poughkeepsie, New York 12601-5400
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements, or other publicly-available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility, or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information
was published. Such trademarks may also be registered or common law trademarks in other countries. A complete and current list of IBM trademarks is available on the Web at:


Adobe is a registered trademark of Adobe Systems Incorporated in the United States, and/or other countries.

Intel is a registered trademark of Intel Corporation in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be the trademarks or service marks of others.
**Glossary**

**access control.** The process of limiting access to system objects and resources to authorized principals.

**access control list.** A list of principals and the type of access allowed to each.

**ACL.** See access control list.

**action.** The part of the event response resource that contains a command and other information about the command.

**attribute.** Attributes are either persistent or dynamic. A resource class is defined by a set of persistent and dynamic attributes. A resource is also defined by a set of persistent and dynamic attributes. Persistent attributes define the configuration of the resource class and resource. Dynamic attributes define a state or a performance-related aspect of the resource class and resource. In the same resource class or resource, a given attribute name can be specified as either persistent or dynamic, but not both.

**AIX.** Advanced Interactive Executive. See AIX operating system.

**AIX operating system.** IBM's implementation of the UNIX operating system.

**authentication.** The process of validating the identity of an entity, generally based on user name and password. However, it does not address the access rights of that entity. Thus, it simply makes sure a user is who he or she claims to be.

**authorization.** The process of granting or denying access to an entity to system objects or resources, based on the entity's identity.

**checksum.** A count of the number of bits in a transmission unit that is included with the unit so that the receiver can check to see whether the same number of bits arrived. If the counts match, it's assumed that the complete transmission was received. TCP and UDP communication layers provide a checksum count and verification as one of their services.

**client.** Client applications are the ordinary user interface programs that are invoked by users or routines provided by trusted services for other components to use. The client has no network identity of its own: it assumes the identity of the invoking user or of the process where it is called, who must have previously obtained network credentials.

**cluster.** A group of servers and other resources that act like a single system and enable high availability and, in some cases, load balancing and parallel processing.

**clustering.** The use of multiple computers (such as UNIX workstations, for example), multiple storage devices, and redundant interconnections to form what appears to users as a single highly-available system. Clustering can be used for load balancing, for high availability, and as a relatively low-cost form of parallel processing for scientific and other applications that lend themselves to parallel operations.

**cluster security services.** A component of RSCT that is used by RSCT applications and other RSCT components to perform authentication within both management domains and peer domains.

**condition.** A state of a resource as defined by the event response resource manager (ERRM) that is of interest to a client. It is defined by means of a logical expression called an event expression. Conditions apply to resource classes unless a specific resource is designated.

**condition/response association.** A link between a condition and a response.

**CSM.** Clusters Systems Management.

**datagram.** Synonymous with UDP packet.

**DNS.** See domain name system.

**domain.** (1) A set of network resources (such as applications and printers, for example) for a group of users. A user logs in to the domain to gain access to the resources, which could be located on a number of different servers in the network. (2) A group of server and client machines that exist in the same security structure. (3) A group of computers and devices on a network that are administered as a unit with common rules and procedures. Within the Internet, a domain is defined by its IP address. All devices that share a common part of the IP address are said to be in the same domain.

**domain name.** A meaningful and easy-to-remember "handle" for an Internet address.

**domain name system.** The service through which domain names are located and translated into IP addresses.

**event.** Occurs when the event expression of a condition evaluates to True. An evaluation occurs each time an instance of a dynamic attribute is observed.

**event expression.** A definition of the specific state when an event is true.

**event response.** One or more actions as defined by the event response resource manager (ERRM) that take place in response to an event or a rearm event.
failover. A backup operation that automatically switches to another adapter if one adapter fails. Failover is an important fault-tolerance function of mission-critical systems that rely on constant accessibility. Automatically and transparently to the user, failover redirects requests from the failed adapter to another adapter that mimics the operations of the failed adapter.

FFDC. See first failure data capture.

first failure data capture. Provides a way to track problems back to their origin even though the source problem may have occurred in other layers or subsystems than the layer or subsystem with which the end user is interacting. FFDC provides a correlator called an ffdc_id for any error that it writes to the AIX error log. This correlator can be used to link related events together to form a chain.

FIFO. First in first out, usually referring to buffers.

High Performance Switch. The switch that works in conjunction with a specific family of IBM System p servers.

HPS. See High Performance Switch.

Internet Protocol. The method by which data is sent from one computer to another on the Internet.

IP. See Internet Protocol.

IP address. A 32-bit (in IP Version 4) or 128-bit (in IP Version 6) number identifying each sender or receiver of information that is sent in packets across the Internet.

LAPI. See low-level application programming interface.

LAPI for Linux. The version of LAPI that runs on the Linux operating system. See also low-level application programming interface.

Linux. A freeware clone of UNIX for 386-based personal computers (PCs). Linux consists of the linux kernel (core operating system), originally written by Linus Torvalds, along with utility programs developed by the Free Software Foundation and by others.

LoadLeveler. A job management system that works with POE to let users run jobs and match processing needs with system resources, in order to make better use of the system.

low-level application programming interface. A low-overhead message-passing protocol that uses a one-sided communication model and active message paradigm to transfer data among tasks. See also LAPI for Linux and RSCT LAPI for AIX. Contrast with PSSP LAPI.

logical unit number. A unique identifier used on a SCSI bus that enables it to differentiate between up to eight separate devices (each of which is a logical unit).

Each LUN is a unique number that identifies a specific logical unit, which may be an end user, a file, or an application program.

LUN. See logical unit number.

management domain. A set of nodes configured for manageability by the Clusters Systems Management (CSM) licensed program. Such a domain has a management server that is used to administer a number of managed nodes. Only management servers have knowledge of the whole domain. Managed nodes only know about the servers managing them; they know nothing of each other. Contrast with peer domain.

Message Passing Interface. A standardized API for implementing the message-passing model.

MPI. See Message Passing Interface.

mutex. See mutual exclusion object.

mutual exclusion object. A program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously. When a program is started, a mutual exclusion object is created with a unique name. After this stage, any thread that needs the resource must lock the mutual exclusion object from other threads while it is using the resource. The mutual exclusion object is set to unlock when the data is no longer needed or the routine is finished.

network credentials. These represent the data specific to each underlying security mechanism.

OSI. Operating system image.

PAC. See privileged attribute certificate.

packet. The unit of data that is routed between an origin and a destination on the Internet or any other packet-switched network.

Parallel Environment. An IBM licensed program that is an execution and development environment for parallel C, C++, and FORTRAN programs. PE also includes tools for debugging, profiling, and tuning parallel programs.

parallel operating environment. An execution environment that smooths the differences between serial and parallel execution. It lets you submit and manage parallel jobs.

Parallel System Support Programs. The IBM Parallel System Support Programs for AIX 5L licensed program is system administration software for the IBM RS/6000® SP system.

PE. See Parallel Environment.

peer domain. A set of nodes configured for high availability by the configuration resource manager. Such a domain has no distinguished or master node. All
nodes are aware of all other nodes, and administrative commands can be issued from any node in the domain. All nodes also have a consistent view of the domain membership. Contrast with management domain.

POE. See parallel operating environment.

port. A "logical connection place". Using TCP/IP, the way a client program specifies a particular server program on a computer in a network.

principal. A user, an instance of the server, or an instance of a trusted client whose identity is to be authenticated.

privileged attribute certificate. Contains such information as the client's name and the groups to which it belongs. Its format is dependent on the underlying security mechanism.

protocol. The set of rules that endpoints in a telecommunication connection use when they communicate.

PSSP. See Parallel System Support Programs.

PSSP LAPI. The version of LAPI that supports the SP Switch2.

rearm event. Occurs when the rearm expression for a condition evaluates to True.

rearm expression. An expression that generates an event which alternates with an original event in the following way: the event expression is used until it is true; then, the rearm expression is used until it is true; then, the event expression is used. The rearm expression is commonly the inverse of the event expression. It can also be used with the event expression to define an upper and lower boundary for a condition of interest.

Reliable Scalable Cluster Technology. A set of software components that together provide a comprehensive clustering environment for AIX, Linux, and Windows. RSCT is the infrastructure used by a variety of IBM products to provide clusters with improved system availability, scalability, and ease of use.

resource. An entity in the system that provides a set of services. Examples of hardware entities are processors, disk drives, memory, and adapters. Examples of software entities are database applications, processes, and file systems. Each resource in the system has one or more attributes that define the state of the resource.

resource class. A broad category of system resource, for example: node, file system, adapter. Each resource class has a container that holds the functions, information, dynamic attributes, and conditions that apply to that resource class. For example, the /tmp space used condition applies to a file system resource class.

resource manager. A process that maps resource and resource-class abstractions into calls and commands for one or more specific types of resources. A resource manager can be a standalone daemon, or it can be integrated into an application or a subsystem directly.

RSCT. See Reliable Scalable Cluster Technology.

RSCT LAPI for AIX. The version of LAPI that runs on the AIX 5.3 and 6.1 operating systems. See also low-level application programming interface.

RSCT peer domain. See peer domain.

SCSI. See Small System Computer Interface.

Small System Computer Interface. A parallel interface that can have up to eight devices all attached through a single cable; the cable and the host (computer) adapter make up the SCSI bus. The bus allows the interchange of information between devices independently of the host. In the SCSI program, each device is assigned a unique number, which is either a number between 0 and 7 for an 8-bit (narrow) bus, or between 8 and 16 for a 16-bit (wide) bus. The devices that request input/output (I/O) operations are initiators and the devices that perform these operations are targets. Each target has the capacity to connect up to eight additional devices through its own controller; these devices are the logical units, each of which is assigned a unique number for identification to the SCSI controller for command processing.

SD. Structured data.

security context token. A pointer to an opaque data structure called the context token descriptor. The context token is associated with a connection between a client and the server.

security services token. A pointer to an opaque descriptor called the security token descriptor. It keeps track of the mechanism-independent information and state.

servers. Server programs are usually daemons or other applications running in the background without a user's inherited credentials. A server must acquire its own network identity to get to access other trusted services.

SP Switch2. The switch that works in conjunction with IBM RS/6000 SP systems.

standalone system. A system on which you are using LAPI for Linux or RSCT LAPI for AIX that is not running PE.
**striping.** The distribution of message data across multiple communication adapters in order to increase bandwidth.

**TCP.** See *Transmission Control Protocol.*

**Transmission Control Protocol.** One of the core Internet protocols. TCP ports are 16-bit entities, so a maximum of 65535 different endpoints are possible within a single IP address.

**UDP.** See *User Datagram Protocol.*

**User Datagram Protocol.** One of the core Internet protocols. UDP is a layer 4 protocol (Transport layer of the OSI model) within the Internet protocol suite. It provides a mechanism to identify different endpoints on a single host by using ports. UDP deals with single-packet delivery that is provided by the underlying IP. As a stateless protocol, it is often used in applications where data must arrive quickly. This smaller feature set provides quicker data transmittal and lower total overhead. UDP packets (or datagrams) contain, in addition to the lower-level headers, a UDP header, which consists of the packet length, source and destination ports, and a checksum. UDP ports are 16-bit entities, so a maximum of 65535 different endpoints are possible within a single IP address.
Index

A
accessibility features 299
active message 73
active messages 20
address-related functions 19
advantages of LAPI 15
atomicity 13
audience of this book xiii

B
bibliography xv
blocking calls 12
books
RSCT xv
buffer
data 11

C
C bindings 86
calls
blocking 12
non-blocking 12
case sensitivity in AIX 302
coexistence 9
commands
LAPI 147
pnsd_stat 148
pnsd_trigger 150
communication behaviors
of LAPI 12
communication operation
completion of 12
completion
of communication operation 12
completion detection 24
constants 333
context, LAPI 11
conventions
terminology xv
typographic xiii
counters 12

D
data buffer 11
data transfer 45
generic 22
non-contiguous 21, 47
strided 51
vector 47
datatypes 329
LAPI_GEN_GENERIC 49
LAPI_GEN_IOVECTOR 50
definitions 341
destination 11

E
environment
querying 18
setting up 18
error handling 13
event notification 24
examples
subroutine 153
execution model
IP/US 303

F
failover 14, 127
feedback
product-related 302
FORTRAN bindings 86
functional overview 17
functions
address-related 19
get 19
pull 19
push 19
put 19
functions of LAPI 17

G
get functions 19
glossary 341

H
handle, LAPI 11
handlers 12

I
initialization 18
instance, LAPI 11
interrupt mode 13
introduction 11
IP/US
execution model 303
ISO 9000 302

L
LAPI
characteristics 11
commands 147
communication behaviors 12
programs
compiling 91
segment registers 335
version 301
LAPI constants 333
LAPI context 11
LAPI datatypes 329
LAPI handle 11
LAPI instance 11
LAPI size limits 333
LAPI_Addr_get subroutine 158
LAPI_Addr_set subroutine 160
LAPI_Address subroutine 162
LAPI_Address_init subroutine 164
LAPI_Address_init64 166
LAPI_Amsend subroutine 168
LAPI_Amsendv subroutine 175
LAPI_Fence subroutine 181
LAPI_free_create subroutine 197
LAPI_GEN_GENERIC 49
LAPI_GEN_IOVECTOR 50
LAPI_Get subroutine 183
LAPI_Getcntr subroutine 186
LAPI_Getv subroutine 188
LAPI_Gfence subroutine 193
LAPI_Group_create subroutine 195
LAPI_Init subroutine 199
LAPI_Msg_string subroutine 206
LAPI_Msgpoll subroutine 208
LAPI_Nopoll_wait subroutine 282
LAPI_Probe subroutine 211
LAPI_Purge_totask subroutine 284
LAPI_Put
  with shared memory 305
  without shared memory 306
LAPI_Put subroutine 213
LAPI_Putv subroutine 216
LAPI_Qenv subroutine 221
LAPI_Resume_totask subroutine 286
LAPI_Rmw subroutine 226
LAPI_Rmw64 subroutine 230
LAPI_Senv subroutine 234
LAPI_Setcntr subroutine 236
LAPI_Setcntr_wstatus subroutine 288
LAPI_Term subroutine 239
LAPI_Util subroutine 241
LAPI_Waitcntr subroutine 258
LAPI_Xfer structure types 261
LAPI_Xfer subroutine 260
lapi_xfer_type_t 261
lock sharing 14, 111

M
message ordering 13
messages 13
migration 9
mpxlf_r 87

N
non-blocking calls 12
non-contiguous data transfer 47

O
operation
  standalone operation 15
operations
  get 19
  pull 12, 19
  push 12, 19
  put 19
  remote read-modify-write 21
origin 11
overview of LAPI 17

P
pnsd_stat command 148
pnsd_trigger command 150
polling mode 13
prerequisite information xv
prerequisite knowledge for this book xiii
product-related feedback 302
profiling 14
  export file 87
  LAPI name-shift 86
  library 87
  shared library 87
  profiling library 86
programs
  compiling 91
  running 91
progress 13, 24
publications
  RSCT xv
pull operations 12, 19
push operations 12, 19
put functions 19

R
RDMA transfer
  user-initiated 14
recovery 14, 127
related information xv
remote read-modify-write operations 21
RSCT
  books xv
  feedback 302
  publications xv
runtime environment
  querying 18
  setting up 18

S
sample subroutine 153
sharing locks 14, 111
size limits 333
source 11
standalone operation 15
statistics 14
strided data transfer 51
striping 14, 127
subroutine sample 153
subroutines 157, 281
  LAPI_Addr_get 158
  LAPI_Addr_set 160
  LAPI_Address 162
  LAPI_Address_init 164
  LAPI_Address_init64 166
  LAPI_Amsend 168
  LAPI_Amsendv 175
  LAPI_Fence 181
  LAPI_Get 183
  LAPI_Getcntr 186
  LAPI_Getv 188
  LAPI_Gfence 193
  LAPI_Group_create 195
  LAPI_Group_free 197
  LAPI_Init 199
  LAPI_Msg_string 206
  LAPI_Msgpoll 208
  LAPI_Nopoll_wait 282
  LAPI_Probe 211
  LAPI_Purge_totask 284
  LAPI_Put 213
  LAPI_Putv 216
  LAPI_Qenv 221
  LAPI_Resume_totask 286
  LAPI_Rmw 226
  LAPI_Rmw64 230
  LAPI_Senv 234
  LAPI_Setcntr 236
  LAPI_Setcntr_wstatus 288
  LAPI_Term 239
  LAPI_Util 241
  LAPI_Waitcntr 258
  LAPI_Xfer 260

T
  target 11
  termination 18
  terminology 341
  terminology conventions xv
  trademarks 338
  transfer
    user-initiated RDMA 14
  types
    LAPI_GEN GENERIC 49
    LAPI_GEN IVECTOR 50
  typographic conventions xiii

U
  user-initiated RDMA transfer 14

V
  vector data transfer 47
    strided 51
  version
    of LAPI 301

Index 347
Readers’ comments – We’d like to hear from you

Reliable Scalable Cluster Technology
LAPI Programming Guide

Publication No. SA22-7936-09

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.
Send your comments to the address on the reverse side of this form.
If you would like a response from IBM, please fill in the following information:

Name
Address
Company or Organization
Phone No.
E-mail address
Readers' Comments — We'd Like to Hear from You

SA22-7936-09

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

IBM Corporation
Department 58HA, Mail Station P181
2455 South Road
Poughkeepsie, NY 12601-5400

POSTAGE WILL BE PAID BY ADDRESSEE